# redis

## redis概述

### reduis的引入

在传统的Java Web项目中，使用数据库进行存储数据，但是有一些致命的弊端，这些弊端主要来自于性能方面。由于数据库持久化数据主要是面向磁盘，而磁盘的读/写比较慢，在一般管理系统上，由于不存在高并发，因此往往没有瞬间需要读/写大量数据的要求，这个时候使用数据库进行读/写是没有太大的问题的，但是在互联网中，往往存在大数据量的需求，比如一些商品抢购的场景，或者是主页讲问量瞬间较大的时候，一瞬间成千上万的

请求就会到来，需要系统在极短的时间内完成成千上万次的读/写操作，这个时候往往不是数据库能够承受的，极其容易造成数据库系统瘫痪，最终导致服务宕机的严重生产问题。为了克服这些问题，Java Web项目往往就引入了NoSQL技术，NoSQL工具也是一种简易的数据库，它主要是一种基于内存的数据库，并提供一定的持久化功能。Redis和MongoDB是当前使用最广泛的NoSQL，而本书主要介绍的是Redis技术，它的性能十分优越，可以支持每秒十几万次的读/写操作，其性能远超数据库，并且支持集群、分布式、

主从同步等配置，原则上可以无限扩展，让更多的数据存储在内存中，而更让我们感到欣喜的是它还能支持一定的事务能力，这在高并发访问的场景下保证数据安全和一致性特别有用。

Redis的性能优越主要来自于3个方面。首先，它是基于ANSIC语言编写的，接近于汇编语言的机器语言，运行十分快速。其次，它是基于内存的读/写，速度自然比数据库的磁盘读/写要快得多。最后，它的数据库结构只有6种数据类型，数据结构比较简单，因此规则较少，而数据库则是范式，完整性、规范性需要考虑的规则比较多，处理业务会比较复杂。所以一般而言Redis的速度是正常数据库的几倍到几十倍，如果把命中率高的数据存储在Redis上，通过Redis读/写和操作这些数据，系统的性能就会远超只使用数据库的情况，所以用好Redis对于Java互联网项目的响应速度和性能是至关重要的。

### redis介绍

Redis:REmote DIctionary Server(远程字典服务器）

属是完全开源免费的，用C语言编写的，遵守BSD协议，是一个高性能的（key/value)分布式内存数据库，基于内存运行并支持持久化的NoSQL数据库，是当前最热门的NoSql数据库之，也被人们称为数据结构服务器。

Redis 与其他key-value缓存产品有以下三个特点。

Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用。

Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储。

Redis支持数据的备份，即master-slave模式的数据备份。

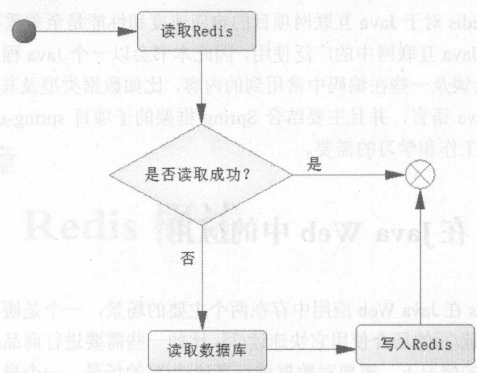
### redis主要作用

1、缓存

在对数据库的读/写操作中，现实的情况是读操作的次数远超写操作，一般是1：9到3:7的比例，所以需要读的可能性是比写的可能性多得多。当发送SQL去数据库进行读取时，数据库就会去磁盘把对应的数据索引回来，而索引磁盘是一个相对缓慢的过程。如果把数据直接放在运行在内存中的Redis服务器上，那么不需要去读/写磁盘了，而是直接读取内存，显然速度会快得多，并且会极大减轻数据库的压力。

而使用内存进行存储数据开销也是比较大的，因为磁盘可以是TGB级别，而且十分廉价，内存一般是几百个GB就相当了不起了，所以内存虽然高效但空间有限，价格也比磁盘高许多，因此使用内存代价较高，并不是想存什么就存什么，因此我们应该考虑有条件的存储数据。一般而言，存储一些常用的数据，比如用户登录的信息；一些主要的业务信息，比如银行会存储一些客户基础信息、银行卡信息、最近交易信息等。一般而言在使Redis 存储的时候，需要从3个方面进行考虑。

业务数据常用吗？命中率如何？如果命中率很低，就没有必要写入缓存。该业务数据是读操作多，还是写操作多，如果写操作多，频繁需要写入数据库，也没有必要使用缓存。业务数据大小如何？如果要存储几百兆字节的文件，会给缓存带来很大的压力，有没有必要？



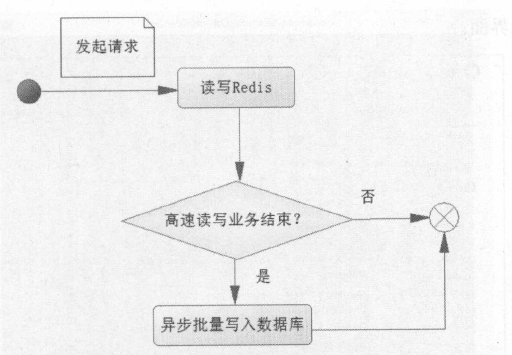
当第一次读取数据的时候，读取Redis的数据就会失败，此时会触发程序读取数据库，把数据读取出来，并且写入Redis。

当第二次及以后读取数据时，就直接读取Redis，读到数据后就结束了流程，这样速度就大大提高了。

2）高速读写场合

在互联网的应用中，往往存在一些需要高速读/写的场合，比如商品的秒杀，抢红包，淘宝、京东的双十一活动或者春运抢票等。这类场合在一个瞬间成千上万的请求就会达到服务器，如果使用的是数据库，一个瞬间数据库就需要执行成千上万的SQL，很容易造成数据库的瓶颈，严重的会导致数据库瘫痪，造成Java Web系统服务崩溃。

在这样的场合的应对办法往往是考虑异步写入数据库，而在高速读/写的场合中单单使用Redis去应对，把这些需要高速读/写的数据，缓存到Redis中，而在满足一定的条件下，触发这些缓存的数据写入数据库中。先看看一次请求操作的流程图，如图17-3所示。



进一步论述这个过程：当一个请求达到服务器，只是把业务数据先在Redis读/写，而没有进行任何对数据库的操作，换句话说系统仅仅是操作Redis缓存，而没有操作数据库，这个速度就比操作数据库要快得多，从而达到需要高速响应的效果。但是一般缓存不能持久化，或者所持久化的数据不太规范，因此需要把这些数据存入数据库，所以在一个请求操作完Redis的读/写后，会去判断该高速读/写的业务是否结束。这个判断的条件住往就是秒杀商品剩余个数为0，抢红包金额为0，如果不成立，则不会操作数据库；如果成立，则触发事件将Redis缓存的数据以批量的形式一次性写入数据库，从而完成持久化的工作。

假设面对的是一个商品秒杀的场景，从上面的流程看，一个用户抢购商品，绝大部分的场合都是在操作内存数据库Redis，而不是磁盘数据库，所以其性能更为优越。只有在商品被抢购一空后才会触发系统把Redis缓存的数据写入数据库磁盘中，这样系统大部分的操作基于内存，就能够在秒杀的场合高速响应用户的请求，达到快速应答。而现实中这种需要高速响应的系统会比上面的分析更复杂，因为这里没有讨论高并发下的数据安全和一致性问题，没有讨论有效请求和无效请求、事务一致性等诸多问题。

### redis与数据库的异同

和数据库一样类，Redis等NoSQL工具也能够存储数据，有人认为NoSQL将来会取代数据库，但是笔者却不那么认为，这里谈谈NoSQL和传统数据库的差异。

首先，NoSQL的数据主要存储在内存中（部分可以持久化到磁盘），而数据库主要是磁盘。其次，NoSQL数据结构比较简单，虽然能处理很多的问题，但是其功能毕竟是有限的，不如数据库的SOL语句强大，支持更为复杂的计算。再次，NoSQL并不完全安全稳定，由于它基于内存，一旦停电或者机器故障数据就很容易丢失数据，其持久化能力也是有限的，而基于磁盘的数据库则不会出现这样的问题。最后，其数据完整性、事务能力、安全性、可靠性及可扩展性都远不及数据库。

基于以上原因，笔者并不认为为NoSQL会取代数据库。毫无疑问，Redis作为一NoSQL是十分成功的，但是它的成功主要是解决互联网系统的一些问题，而主要的问题是性能问题。实际上，在互联网系统大部分的业务场景中，业务都是相对简单的，而难以处理的问题主要是性能问题，特别是那些会员数比较多的高并发服务网站。例如，你可以常常在淘宝或者京东网站上看到一个即将被抢购的商品，有多达几万人的关注，可能一个时刻就发生了成千上万笔业务，此时使用Redis作为缓存数据，就可以明显提升系统的性能，而且这十分有效。所以基于两者之间的区别，笔者认为使用NoSQL去取代数据库，目前还做不到，但是作为一种提高互联网应用性能的辅助工具，它十分有用。

### Redis的高并发

官方的bench-mark数据:测试完成了50个并发执行100000请求。设置和获取的值是一个256字节字符串。结果:读的速度是110000次/s,写的速度是81000次/s. redis尽量少写多读,符合缓存的适用要求。单机redis支撑万级,如果10万+可以采用主从复制的模式。

1, Redis是纯内存数据库,所以读取速度快。

2, Redis使用的是非阻塞10, 10多路复用,减少了线程切换时上下文的切换和竞争。

3. Redis采用了单线程的模型,保证了每个操作的原子性,也减少了线程的上下文切换和竞争。

4, Redis存储结构多样化,不同的数据结构对数据存储进行了优化加快读取的速度。

5. Redis采用自己实现的事件分离器,效率比较高,内部采用非阻塞的执行方式,吞吐能力比较大。

## redis下载安装

### 下载

<https://redis.io/download>

### 解压安装

创建目录：mkdir /usr/local/redis

解压 ：tar -zxvf redis.tar

### 编译

先安装 gcc-c++:yum -y install gcc-c++

的解压目录里面:cd regdis

编译:make

### 安装

安装目录：make install PREFIX=/usr/local/redis

### 安装后的目录

只有一个bin目录



redis-benchmark：性能测试工具

redis-check-aof：修复有问题的AOF文件

redis-check-dump：修复有问题的dump.rdb文件

redis-cli：客户端，操作入口

redis-sentinel：redis集群使用

redis-server：Redis服务器启动命令

### 启动redis

./redis-server

新开一个桌面连接redis

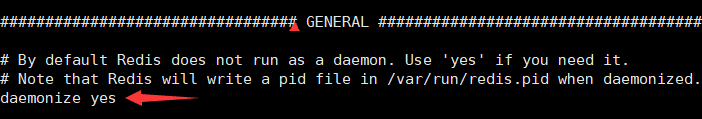
./redis-cli

### 自定义redis的配置文件

找到redis解压目录种的配置文件:redis.conf

赋值到bin目录下：cp redis.conf ../bin/

修改redis.cnf的general(一般配置)的daemonize 为yes，默认为非后台配置，修改为yes后，后台启动。



### 查看redis服务是否启动

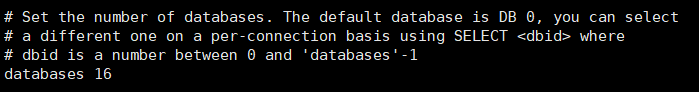
ps -ef|grep redis

验真是否启动：redis-cli -p 6379 登录

发送:ping 他会返回pong



### 查看redis的数据库



redis的默认数据库为16个，0-15，可以通过select n 来切换数据库。

例如：切换到6库

select 7

### Docker安装redis

## 常用命令

1、切换数据库

select n

注：默认为16个，从0开始

2、查看当前数据库的key的数量

Dbsize

3、查看所有的键

keys \*

4、清空库或清空所有库 flushdb：删除当前库

flushall：删除所有库

### redis的配置文件

|  |
| --- |
|  |

## Redis数据类型分析

### key

1)keys \* 查看所有的键

2)exists key 名字，判断某个key是否存在,

存在：返回1

不存在：返回0

3)move key db :当前库就没有了，一出了

4)expire key 秒钟：为给定的key设置过期时间

5)ttl key 查看还有多少秒过期，-1表示永不过期-2表示已经过期

已经过期后，将会移除系统

6)type key 查看你的key是什么类型。

7)如果设置相同的key，后一个插入的key会覆盖前面的key。

### 字符串(string)

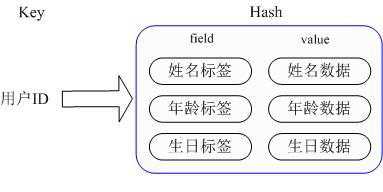
字符串是Redis最简单的储存类型,它存储的值可以是字符串、整数或者浮点数,对整个字符串或者字符串的其中一部分执行操作;对整数或者浮点数执行自增(increment)或者自减(decrement)操作。Redis的字符串是一个由字节组成的序列,跟java里面的Arraylist有点类似,采用预分配冗余空间的方式来减少内存的频繁分配,内部为当前字符串实际分配的空间capacity一般要高于实际字符串长度len,当字符串长度小于1M时,扩容都是加倍现有的空间,如果超过1M,扩容时一次只会多扩1M的空间。需要注意的是字符串最大长度为512M.

常用命令：

|  |  |
| --- | --- |
| get key | 获取给定键的值 |
| set key | 设置给定键的值 |
| strlen key | 返回给定键的值的长度 |
| incrby key-name amount | 将键存储的值加上整数amont，如果没有mount，默认为加1 |
| decrby key-name amount | 将键存储的值减去整数amont，如果没有mount，默认为减1 |
| getrange | getrange key-name start end，获取一个偏移量start至偏移量end范围内所有字符组成的子串，包括start和end在内 |
| setrange | setrange key-name offset value，将从start偏移量开始的子串设置为给定值 |
| setex key second value | 设置过期时间  setex k4 10 v4 设置键k4生命名为10秒  结果:10秒钟之后返回nil 相当于0 |
| SETNX key value  （SET if Not eXists） | 将 key 的值设为 value ，当且仅当 key 不存在。  若给定的 key 已经存在，则 SETNX 不做任何动作。 |
| mset key1 value1  key2 value2 …… | 同时设置多个值 |
| mget key1 key2 | 同时获取多个值 |
|  |  |

### hash(哈希)

散列相当于Java中的HashMap,内部是无序字典。实现原理跟HashMap一致。一个哈希表有多个节点,每个节点保存一个键值对。与Java中的HashMap不同的是, rehash的方式不一样,因为Java的HashMap在字典很大时, rehash是个耗时的操作,需要一次性全部rehash, Redis为了高性能,不能堵塞服务,以采用了渐进式rehash策略。渐进式rehash会在rehash的同时,保留新旧两个hash结构,查询时会同时查询两个hash结构,然后在后续的定时任务中以及hash操作指令中,循序渐进地将旧hash的内容一点点迁移到新的hash结构中。当搬迁完成了,就会使用新的hash结构取而代之。当hash移除了最后一个元素之后,该数据结构自动被删除,内存被回收。

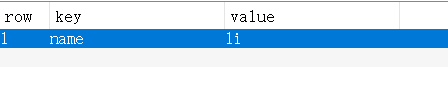


常用命令：

（1）hset——设置key对应的HashMap中的field的value

格式：hset key field value

例如：hset hash name li



（2）hget——获取key对应的HashMap中的field的value

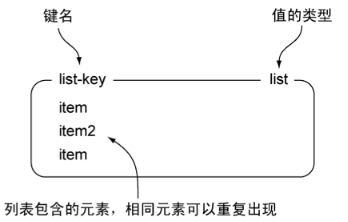
格式：hget key field

（3）hgetall——获取key对应的HashMap中的所有field的value

格式：hgetall key



### List结构



Redis列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素导列表的头部（左边)或者尾部（右边）。它的底层实际是个链表。

Redis的List类型其实就是每一个元素都是String类型的双向链表。我们可以从链表的头部和尾部添加或者删除元素。这样的List既可以作为栈，也可以作为队列使用。

特点:

1、如果间不存在，则创建新的链表

2、如果键已经存在，新增内容

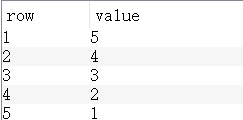
3、如果值全部移除，对应的键也消失了

4、内部使用双向链表实现，所以获取越接近两端的元素速度越快，但通过索引访问时会比较慢

常用命令

添加左边元素：LPUSH 语法：LPUSH key value [value ...] ，返回添加后的列表元素的总个数

LPUSH list 1 2 3 4 5



添加右边元素：RPUSH 语法：RPUSH key value [value ...] ，返回添加后的列表元素的总个数



移除左边第一个元素：LPOP 语法：LPOP key ，返回被移除的元素值

移除右边第一个元素：RPOP 语法：RPOP key ，返回被移除的元素值

获取list中的值 ：lrange 语法：lrange key start end

例如：lrange key 0 -1 全取key的值

列表元素个数：LLEN 语法：LLEN key， 不存在时返回0，redis是直接读取现成的值，并不是统计个数

按照索引下标回去元素(从上到下)： 语法：lindex key

删除N个value 语法：lrem key 删除所有key 的值

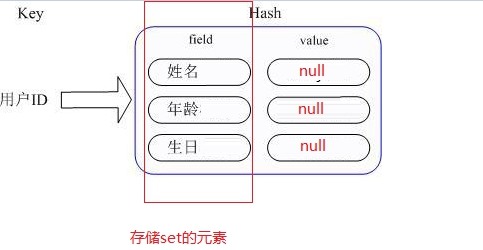
应用场景:

lists的应用场景非常多,可以利用它轻松实现热销榜;可以实现工作队列(利用lists的push操作,将任务存在lists中,然后工作线程再用pop操作将任务取出进行执行) ;可以实现最新列表,比如最新评论等。

### set

Redis 集合（Set类型）是一个无序的String类型数据的集合，类似List的一个列表，与List不同的是Set不能有重复的数据。实际上，Set的内部是用HashMap实现的，Set只用了HashMap的key列来存储对象。

可见创建一个HashSet的时候实际上创建了一个HashMap；Set中的元素，只是存放在了底层HashMap的key上，底层HashMap的value列为空，遍历HashSet的时候从HashMap中取出keySet来遍历。



常用命令

（1）sadd——在key对应的set中添加一个元素。

（2）smembers——获取key对应的set的所有元素。

（3）spop——随机返回并删除key对应的set中的一个元素



应用场景: redis的sets类型是使用哈希表构造的,因此复杂度是0(1),它支持集合内的增删改查,并且支持多个集合间的交集、并集、差集操作。可以利用这些集合操作,解决程序开发过程当中很多数据集合间的问题。比如计算网站独立ip,用户画像中的用户标签,共同好友等功能

### Sortset

SortSet顾名思义，是一个排好序的Set，它在Set的基础上增加了一个顺序属性score，这个属性在添加修改元素时可以指定，每次指定后，SortSet会自动重新按新的值排序。

sorted set的内部使用HashMap和跳跃表(SkipList)来保证数据的存储和有序，HashMap里放的是成员到score的映射，而跳跃表里存放的是所有的成员，排序依据是HashMap里存的score。

常用命令

（1）zadd ——在key对应的zset中添加一个元素

格式：zadd key sort value sort value sort value

例如zadd zset 1 li 2 sx 3 lw

（2）zrange——获取key对应的zset中指定范围的元素，-1表示获取所有元素

获取全部元素：zrange myzset 0 -1

3、zrem——删除key对应的zset中的一个元素

例如：zrem myzset li 删除li这个元素



主要应用于根据某个权重进行排序的队列的场景,比如游戏积分排行榜,设置优先级的任务列表,学生成绩表等。

### sorted 数据结构

跳跃表

## ziplist

### 引入

为什么要有ziplist?

1、普通的双向链表，会有两个指针，在存储数据很小的情况下，我们存储的实际数据的大小可能还没有指针占用的内存大，是不是有点得不偿失？而且Redis是基于内存的，而且是常驻内存的，内存是弥足珍贵的，所以Redis的开发者们肯定要使出浑身解数优化占用内存，于是，ziplist出现了。

2、链表在内存中，一般是不连续的，遍历相对比较慢，而ziplist可以很好的解决这个问题。

### Ziplist

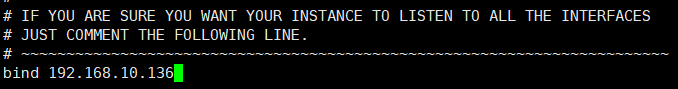
在Redis中，有五种基本数据类型，除了 String，还有list，hash，zset，set，其中list，hash，zset都间接或者直接使用了ziplist，所以说理解ziplist也是相当重要的。

我刚开始看ziplist的时候，总觉得zip这个单词甚是熟悉，好像在日常使用电脑的时候经常看到，就是“压缩”的意思，那ziplist就可以翻译成“压缩列表”了。

## Java 访问Redis

### Redis 外网访问

1)修改配置文件redis.conf

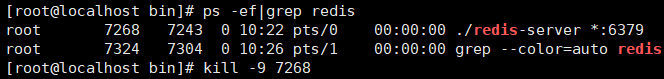


修改127.0.0.7为linux的访问ip

2)修改成功之后重启服务器，在redis 里面没有重启的命令，我们可以通过杀死进程，然后，在启动。

查看当前的进程:ps -ef|grep redis

强制杀死进程：kill -9 进程号



3)再次启动redis-server(加载配置文件)

命令 ./redis-server redis.conf

4）也可以使用redis-cli直接连接

./redis-cli -h 192.168.10.136 -p 6379

## java 连接redis

### 使用工具:jedis maven依赖

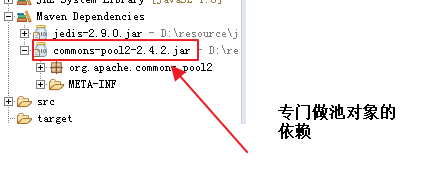
|  |
| --- |
| <properties>  <redis.version>2.9.0</redis.version>  </properties>  <dependencies>  <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  <version>${redis.version}</version>  </dependency>  </dependencies> |

测试代码

|  |
| --- |
| **public** **class** App {  **public** **static** **void** main(String[] args) {  // 新建一个jedis的客户端  String host = "192.168.10.136";  Jedis jedis = **null** ;  **try** {  jedis = **new** Jedis(host);  // jedis.set("java", "java");  String value = jedis.get("java");  // System.out.println("set ok");  System.***out***.println("value:"+value);  }**catch** (Exception e) {  e.printStackTrace();  }**finally** {  // 关闭资源的操作  **if**(**null**!=jedis) {  **try** {  jedis.close();  }**catch** (Exception e) {  e.printStackTrace();  }**finally** {  jedis = **null** ;  }  }  }  }  } |

## Redis的连接池管理redis

连接是一个对象，对象池（commons-pool 2apach的工具）





Method(Jedis): Close()

|  |
| --- |
| **public** **class** app {  **private** **static** JedisPoolConfig *poolConfig* = **new** JedisPoolConfig();  **private** **static** JedisPool *pool* = **null**;  **private** **static** **final** String ***host*** = "192.168.10.136";  **static** {  *poolConfig*.setMaxTotal(20);//连接池最大数量  *poolConfig*.setMaxIdle(15); //连接池最大空闲数量  *poolConfig*.setMinIdle(10); //连接池最小空闲数量  *pool* = **new** JedisPool(*poolConfig*,***host***);  }  **public** **static** **void** main(String[] args) {  Jedis jedis = *pool*.getResource();  String value = jedis.get("sx");  System.***out***.println(value);  //连接池归还或者是关闭  jedis.close();  }  } |

## RidesTemplate

由于Redis只能提供基于字符串型的操作，而在Java中使用的却以类对象为主，所以

需要Redis存储的字符串和Java对象相互转换。如果自己编写这些规则，工作量还是比较大的，比如一个角色对象，我们没有办法直接把对象存入Redis中，需要进一步进行转换，所以对操作对象而言，使用Redis还是比较难的。好在Spring对这些进行了封装和支持，它提供了序列化的设计框架和一些序列化的类，使用后它可以通过序列化把Java对象转换，

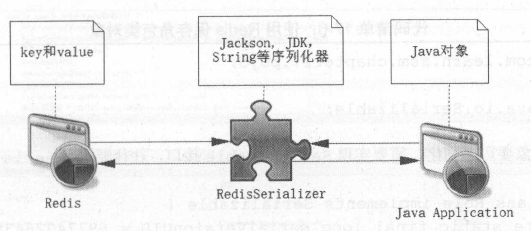
使得Redis能把它存储起来，并且在读取的时候，再把由序列化过的字符串转化为Java对象，这样在Java环境中使用Redis就更加简单了，所以更多的时候可以使用Spring提供Redis Template的机制来使用Redis。

普通的连接使用没有办法把Java对象直接存入Redis，而需要我们自己提供方案，这

时往往就是将对象序列化，然后使用Redis进行存储，而取回序列化的内容后，在通过转

换转变为Java对象，Spring模板中提供了封装的方案，在它内部提供了RedisSerializer接

口（org.springframework.data.redis.serializer.RedisSerializer)和一些实现类。



GenericJackson2JsonRedisSerializer,通用的使用Json2.jar的包，将Redis对象的序列化器。

Jackson2JsonRedisSerializer<T>,通过Jackson2.jar包提供的序列化进行转换。

JdkSerializationRedisSerializer<T>,使用JDK的序列化器进行转化。

OxmSerializer，使用Spring O/X对象Object和XML相互转换

StringRedisSerializer,使用字符串进行序列化。

Generic ToStringSerializer,通过通用的字符串序列化进行相互转换。

使用它们就能够帮助我们把对象通过序列化存储到Redis中，也可以把Redis存储的内容转换为Java对象，为此Spring提供的Redis Template还有两个属性。

## Jedis和Spring 整合

### 添加依赖

|  |
| --- |
| <properties>  <spring-context.version>4.3.20.RELEASE</spring-context.version>  <jedis.version>2.9.0</jedis.version>  </properties>  <dependencies>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  <version>${spring-context.version}</version>  </dependency>  <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  <version>${jedis.version}</version>  </dependency>  </dependencies> |

### spring-jedis.xml配置文件

//配置连接池

|  |
| --- |
| <bean id=*"poolConfig"* class=*"redis.clients.jedis.JedisPoolConfig"*>  <property name=*"maxTotal"* value=*"20"*></property>  <property name=*"maxIdle"* value=*"15"*></property>  <property name=*"minIdle"* value=*"10"*></property>  </bean>  <bean id=*"pool"* class=*"redis.clients.jedis.JedisPool"*>  <constructor-arg name=*"host"* value=*"192.168.10.136"*></constructor-arg>  <constructor-arg name=*"poolConfig"* ref=*"poolConfig"*></constructor-arg>  </bean> |

app测试

|  |
| --- |
| **public** **static** **void** main(String[] args) {  ApplicationContext context = **new** ClassPathXmlApplicationContext("classpath:spring-jedis.xml");  JedisPool jedisPool =(JedisPool) context.getBean("pool");  Jedis jedis = jedisPool.getResource();  String value = jedis.get("sx");  System.***out***.println(value);//li  jedis.close();  } |

### 软编码(提取配置文件)

jedis.properties

|  |
| --- |
| jedis.host=192.168.10.136  jedis.max.total=20  jedis.max.idle=15  jedis.min.idle=20 |

xml中加载配置文件(namespace中添加约束)

|  |
| --- |
| <context:property-placeholder location=*"classpath:/properties/\*.properties"* />  <bean id=*"poolConfig"* class=*"redis.clients.jedis.JedisPoolConfig"*>  <property name=*"maxTotal"* value=*"${jedis.max.total}"*></property>  <property name=*"maxIdle"* value=*"${jedis.max.idle}"*></property>  <property name=*"minIdle"* value=*"${jedis.min.idle}"*></property>  </bean>  <bean id=*"pool"* class=*"redis.clients.jedis.JedisPool"*>  <constructor-arg name=*"host"* value=*"${jedis.host}"*></constructor-arg>  <constructor-arg name=*"poolConfig"* ref=*"poolConfig"*></constructor-arg>  </bean> |

## Redis和Spring data整合

## Redis和Spring boot 整合(基础版本)

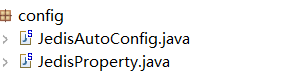
### 创建项目并添加依赖

版本：1.5.18

依赖web，jedis

|  |
| --- |
| <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  </dependency> |

### 配置类文件



JedisProperty

|  |
| --- |
| @Configuration  @ConfigurationProperties(prefix="jedis")  **public** **class** JedisProperty {  **private** String host;  **private** Integer maxTotal;  **private** Integer maxIdle;  **private** Integer minIdle;  } |

JedisAutoConfig

|  |
| --- |
| @Configuration  **public** **class** JedisAutoConfig {  @Autowired  **private** JedisProperty jedisProperty;  @Bean("jedisPool")  **public** JedisPool pool(){  JedisPoolConfig poolConfig = **new** JedisPoolConfig();  poolConfig.setMaxIdle(jedisProperty.getMinIdle());  poolConfig.setMaxTotal(jedisProperty.getMaxTotal());  poolConfig.setMinIdle(jedisProperty.getMinIdle());  **return** **new** JedisPool(poolConfig,jedisProperty.getHost());  }  } |

yum

|  |
| --- |
| jedis:  host: 192.168.10.136  max-total: 20  max-idle: 15  min-idle: 10 |

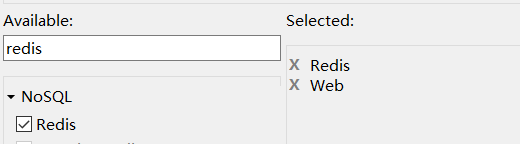
### 测试

|  |
| --- |
| @SpringBootApplication  @RestController  **public** **class** RedisSpringbootApplication {  @Autowired  **private** JedisPool jedisPool;    @RequestMapping("index")  **public** String getInfo() {  Jedis jedis = jedisPool.getResource();  **return** jedis.get("sx");  }  } |

## Spring boot 整合redis(进阶版)

### 创建项目

加载redis的依赖



### 配置文件

yum

|  |
| --- |
| spring:  redis:  host: 192.168.10.136  pool:  max-active: 25  max-idle: 15  min-idle: 10 |

## redis操作

|  |
| --- |
| **public** **class** TestRedis {  ApplicationContext context = **new** ClassPathXmlApplicationContext("classpath:spring-jedis.xml");  JedisPool jedisPool =(JedisPool) context.getBean("pool");  Jedis jedis = jedisPool.getResource();  //1、普通的字符串操作  @Test  **public** **void** test1() {  jedis.set("sx", "li");  String value = jedis.get("sx");  System.***out***.println(value);  }    //2、字节操作  @Test  **public** **void** test2() {  //jedis.set("byte".getBytes(), "byte1".getBytes());  String value = jedis.get("byte");  System.***out***.println(value);//byte1  //对象转为字节  User user = **new** User("sx", "18", "女");  jedis.set("user1".getBytes(), SerializableUtils.*serializable*(user));  }      //3、json的序列化对象  @Test  **public** **void** test3() {  User userjson = **new** User("lee", "22", "男");  jedis.set("userjson", JSON.*toJSONString*(userjson));  }    //4、hash的操作  @Test  **public** **void** test4() {  //写入  //jedis.hset("man", "name", "sx");  //jedis.hset("man", "age", "12");  //获取单个值  String name = jedis.hget("man","name");  String age = jedis.hget("man","age");  System.***out***.println(name+"+"+age);//sx+12  //获取键的全部值  Map<String, String> man = jedis.hgetAll("man");  System.***out***.println(man);  }  //zset集合  @Test  **public** **void** test5() {  //写入  /\*jedis.zadd("pad", 1000,"ls");  jedis.zadd("pad", 2000,"sx");  jedis.zadd("pad", 3000,"df");  jedis.zadd("pad", 4000,"ad");  jedis.zadd("pad", 5100,"df");\*/  //从小到大获取值,获取前三个值  Set<String> pad = jedis.zrange("pad", 0, 2);  **for** (String str : pad) {  System.***out***.println(str);  }  //从大到小获取值,获取前三个值  Set<String> pad1 = jedis.zrevrange("pad", 0, 2);  **for** (String str : pad1) {  System.***out***.println(str);  }  }  } |

## 在spring 里面利用Redis 实现缓存

### 使用切面实现缓存

切面：环绕通知

添加aspect 依赖

|  |
| --- |
| <dependency>  <groupId>org.aspectj</groupId>  <artifactId>aspectjweaver</artifactId>  <version>1.9.2</version>  </dependency> |

切面类

|  |
| --- |
| @Component  @Aspect  **public** **class** CacheAspect {  @Around("execution (\* com.sxt.service.impl.UserServiceImpl.\*(..))")  **public** Object aroud(ProceedingJoinPoint point) **throws** Throwable {  System.***out***.println("方法执行前");  Object result = point.proceed(point.getArgs());  System.***out***.println("方法执行后");  **return** result;  }  } |

配置：开启aop

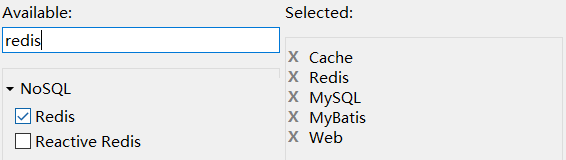
|  |
| --- |
| <!--开启aop-->  <aop:aspectj-autoproxy expose-proxy=*"true"*></aop:aspectj-autoproxy> |

## Spring boot data项目(自带缓存)

添加缓存的注解

|  |  |
| --- | --- |
| @CacheConfig | 提取相同配置的 |
| @Cacheable | 给方法加缓存 |
| @CacheEvient | （删除缓存的值） |
| @Cacheput | （更新缓存的值） |

### 新建项目并选择依赖



依赖：

|  |
| --- |
| <!-- 分页插件的依赖 -->  <dependency>  <groupId>com.github.pagehelper</groupId>  <artifactId>pagehelper-spring-boot-starter</artifactId>  <version>1.2.5</version>  </dependency> |

### 配置文件

|  |
| --- |
| spring:  datasource:  driver-class-name: com.mysql.cj.jdbc.Driver  url: jdbc:mysql://192.168.10.136:3303/hm  username: root  password: 123456  redis:  host: 192.168.10.136  port: 6379  jedis:  pool:  max-active: 25  max-idle: 15  min-idle: 10  mybatis:  mapper-locations:  - classpath:/mapper/\*.xml  #pagehepler配置  pagehelper:  helper-dialect: mysql |

service

|  |
| --- |
| **public** **interface** UserService {    List<User> findAll(**int** page,**int** size);  User findById(Integer id);  User saveUser(User user);  User updateUser(User user);    **void** deleteUser(User user);  **void** deleteUser(Integer id);  } |

serviceimpl

|  |
| --- |
|  |

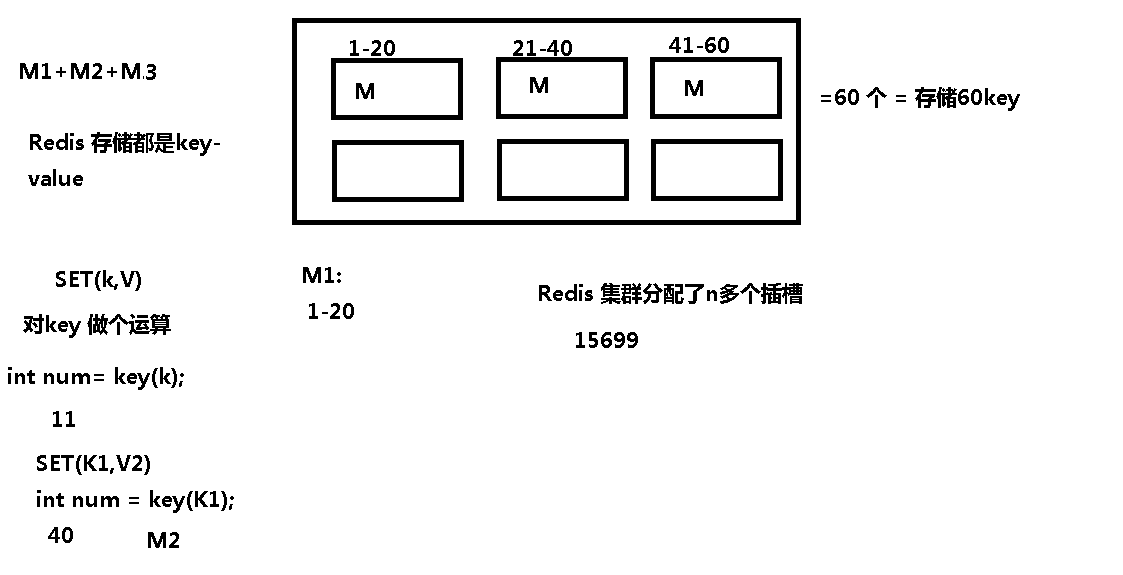
controller

|  |
| --- |
| @RestController  @RequestMapping("api/user")  **public** **class** UserController {  @Autowired  **private** UserService userService;  @PostMapping("/")  **public** String saveUser(User user) {  userService.saveUser(user);  **return** "ok";  }  @DeleteMapping("/{id}")  **public** String deletUser(@PathVariable("id")Integer id) {  userService.deleteUser(id);  **return** "ok";  }  @PutMapping("/")  **public** String updateUser(User user) {  userService.updateUser(user);  **return** "ok";  }  @GetMapping("/{id}")  **public** User findUser(@PathVariable("id")Integer id) {  **return** userService.findById(id);  }  @GetMapping("/")  **public** List<User> findUsers(@RequestParam(defaultValue="1")Integer page,@RequestParam(defaultValue="10")Integer size){  **return** userService.findAll(page, size);  }  } |

启动类

|  |
| --- |
| @SpringBootApplication  @MapperScan("com.sxt.mapper")  @EnableCaching //这里开启缓存  **public** **class** RedisSpringbootDataRedisApplication {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(RedisSpringbootDataRedisApplication.**class**, args);  }  } |

## Redis伪集群的搭建



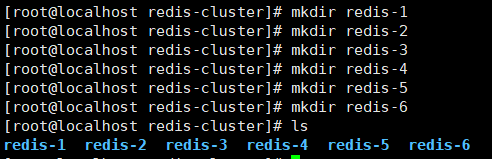
### 伪集群规划

|  |  |  |  |
| --- | --- | --- | --- |
| 文件夹 | Ip | 端口 | 角色 |
| Redis-1 | 192.168.10.136 | 7001 |  |
| Redis-2 | 192.168.10.136 | 7002 |  |
| Redis-3 | 192.168.10.136 | 7003 |  |
| Redis-4 | 192.168.10.136 | 7004 |  |
| Redis-5 | 192.168.10.136 | 7005 |  |
| Redis-6 | 192.168.10.136 | 7005 |  |

### 创建文件夹

1)在/usr/local下创建redis.cluster文件夹。

2)在redis.cluster文件家中创建redis-[1-6]



3)将redis的服务器文件redis-server复制到集群文件夹中。

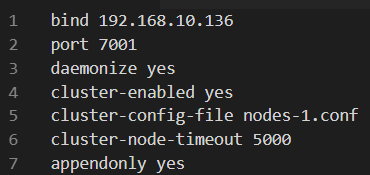
cp redis-server /usr/local/redis-cluster/

### 准备配置文件

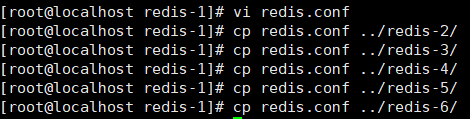
最小化的配置文件

|  |
| --- |
| bind ip  port 7000 端口  daemonize yes 后台运行  cluster-enabled yes 开启集群  cluster-config-file nodes.conf 不用你写，只有集群启动，该文件自动生成，但是该文件必须不同才行  cluster-node-timeout 5000 集群里面的机器互相ping，超时时间  appendonly yes # 开启redis的持久化的功能 |

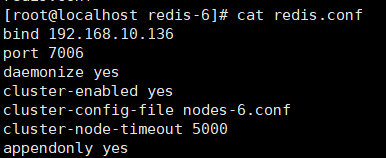
redis-1



复制到其他文件夹中



redis-2

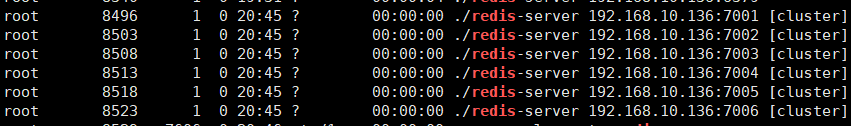


### 开启redis

使用各自的配置文件启动redis

|  |
| --- |
| ./redis-server ./redis-1/redis.conf.  ./redis-server ./redis-2/redis.conf.  ./redis-server ./redis-3/redis.conf.  ./redis-server ./redis-4/redis.conf.  ./redis-server ./redis-5/redis.conf.  ./redis-server ./redis-6/redis.conf. |

查看机器启动状态:ps -ef | grep redis

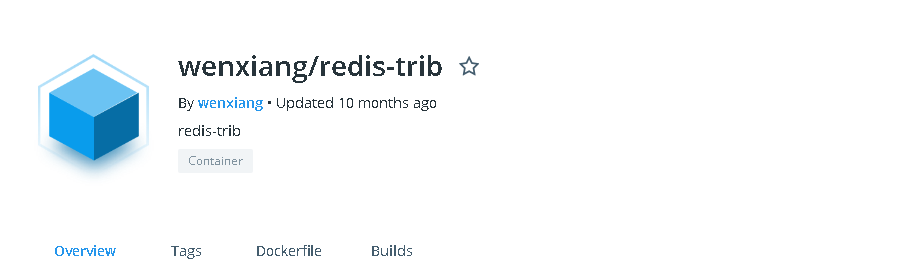


### 使用Ruby脚步搭建

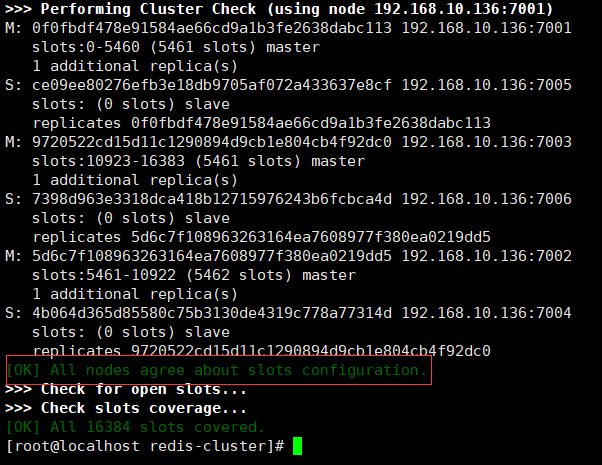
官方写了一个脚本，来配置到底哪个是主机，哪个是从机，需要我们运行一下。

曲线运行脚本，使用docker 运行该脚本

下载镜像：



|  |
| --- |
| docker run -t -i wenxiang/redis-trib create --replicas 1 192.168.10.136:7001 192.168.10.136:7002 192.168.10.136:7003 192.168.10.136:7004 192.168.10.136:7005 192.168.10.136:7006 |



### 连接redis集群



### 普通java项目连接集群

|  |
| --- |
| **public** **class** app {  **private** **static** JedisCluster *jedisCluster* = **null**;  **private** **static** JedisPoolConfig *poolConfig* = **new** JedisPoolConfig();  **private** **static** Set<HostAndPort> *nodes* = **new** HashSet<HostAndPort>();  **static** {  *poolConfig*.setMaxTotal(20);//连接池最大数量  *poolConfig*.setMaxIdle(15); //连接池最大空闲数量  *poolConfig*.setMinIdle(10); //连接池最小空闲数量  *nodes*.add(**new** HostAndPort("192.168.10.136", 7001));  *nodes*.add(**new** HostAndPort("192.168.10.136", 7002));  *nodes*.add(**new** HostAndPort("192.168.10.136", 7003));  *nodes*.add(**new** HostAndPort("192.168.10.136", 7004));  *nodes*.add(**new** HostAndPort("192.168.10.136", 7005));  *nodes*.add(**new** HostAndPort("192.168.10.136", 7006));  *jedisCluster* = **new** JedisCluster(*nodes*,*poolConfig*);  }  **public** **static** **void** main(String[] args) **throws** IOException {  *jedisCluster*.set("sx", "123456");  String value = *jedisCluster*.get("sx");  System.***out***.println(value);  //连接池归还或者是关闭  *jedisCluster*.close();  }  } |

### 在spring中连接redis集群

配置文件

|  |
| --- |
| <bean id=*"poolConfig"* class=*"redis.clients.jedis.JedisPoolConfig"*>  <property name=*"maxTotal"* value=*"${jedis.max.total}"*></property>  <property name=*"maxIdle"* value=*"${jedis.max.idle}"*></property>  <property name=*"minIdle"* value=*"${jedis.min.idle}"*></property>  </bean>  <!--连接单个redis -->  <!-- <bean id="pool" class="redis.clients.jedis.JedisPool">  <constructor-arg name="host" value="${jedis.host}"></constructor-arg>  <constructor-arg name="poolConfig" ref="poolConfig"></constructor-arg>  </bean> -->  <!--连接redis集群 -->  <bean id=*"jedisCluster"* class=*"redis.clients.jedis.JedisCluster"*>  <constructor-arg name=*"poolConfig"* ref=*"poolConfig"*></constructor-arg>  <constructor-arg name=*"nodes"* ref=*"nodes"*></constructor-arg>  </bean>  扫描配置文件  <context:property-placeholder location=*"classpath:/properties/\*.properties"* /> |

配置类，获取配置信息

|  |
| --- |
| @Configuration  **public** **class** JedisClusterNodes {  @Value("${redis.node1}") // ip:port  **private** String node1;  @Value("${redis.node2}")  **private** String node2;  @Value("${redis.node3}")  **private** String node3;  @Value("${redis.node4}")  **private** String node4;  @Value("${redis.node5}")  **private** String node5;  @Value("${redis.node6}")  **private** String node6;  @Bean("nodes") // 将对象放入容器里面  **public** HashSet<HostAndPort> nodes() {  HashSet<HostAndPort> nodes = **new** HashSet<HostAndPort>();  nodes.add(**new** HostAndPort(node1.split(":")[0], Integer.*valueOf*(node1.split(":")[1])));  nodes.add(**new** HostAndPort(node2.split(":")[0], Integer.*valueOf*(node2.split(":")[1])));  nodes.add(**new** HostAndPort(node3.split(":")[0], Integer.*valueOf*(node3.split(":")[1])));  nodes.add(**new** HostAndPort(node4.split(":")[0], Integer.*valueOf*(node4.split(":")[1])));  nodes.add(**new** HostAndPort(node5.split(":")[0], Integer.*valueOf*(node5.split(":")[1])));  nodes.add(**new** HostAndPort(node6.split(":")[0], Integer.*valueOf*(node6.split(":")[1])));  **return** nodes;  }  } |

redis.properties

|  |
| --- |
| redis.node1=192.168.10.136:7001  redis.node2=192.168.10.136:7002  redis.node3=192.168.10.136:7003  redis.node4=192.168.10.136:7004  redis.node5=192.168.10.136:7005  redis.node6=192.168.10.136:7006  jedis.max.total=25  jedis.max.idle=15  jedis.min.idle=10 |

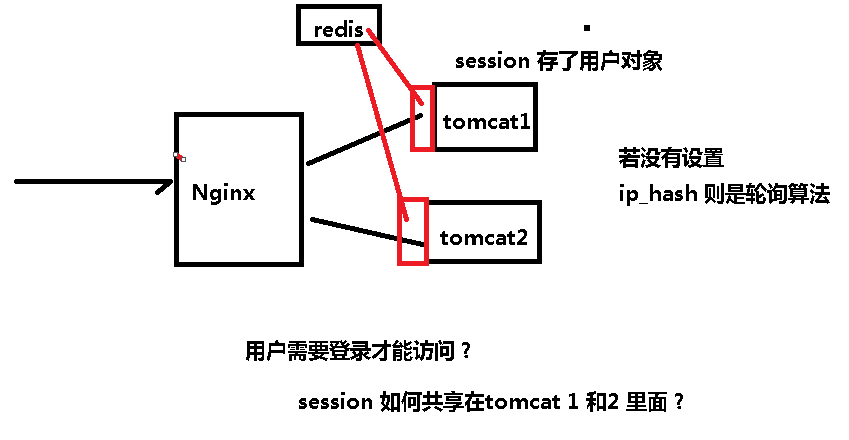
### 在springboot中连接集群

|  |
| --- |
| spring:  redis:  cluster: 修改yum配置文件  nodes:  - 192.168.10.136:7001  - 192.168.10.136:7002  - 192.168.10.136:7003  - 192.168.10.136:7004  - 192.168.10.136:7005  - 192.168.10.136:7005  jedis:  pool:  max-active: 25  max-idle: 15  min-idle: 10 |

## 项目里面redis的使用

### session的存储

用户登录的session共享



### redis做缓存使用

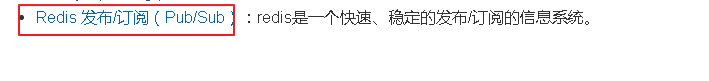
菜单数据

权限数据

### 购物车redis的实现

### 发布订阅功能（redis的发布订阅太简单了，不能用）

Mq解决发布订阅。



## redis的问题

### 删除key的策略

Redis 内存满了删除key的策略

Allkey-lru： 所有的key 都可以参与删除（推荐）

VoliteKey 只有设置过期的key 参与删除

Redis 过期删除如何实现？

立即删除：cpu

过期就删除：

Set(key,value)

给这个key设置过期时间。类似给key有个定时器

惰性删除：耗费内存

给key设置过期时间，key过期了，它并不会删除

当再次访问该key时，才触发删除

定期删除：折中的方案(该方案是默认的)

Redis 一段时间检查所有设置了过期时间的key，若过期，删除

## redis的持久化

### 持久化

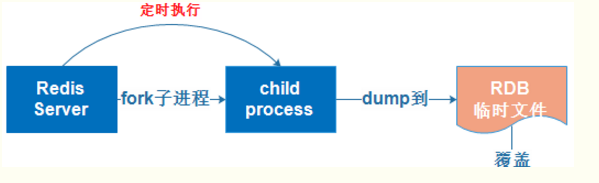
Redis是一种高级key-value数据库。它跟memcached类似，不过数据可以持久化，而且支持的数据类型很丰富。有字符串，链表，集 合和有序集合。支持在服务器端计算集合的并，交和补集(difference)等，还支持多种排序功能。所以Redis也可以被看成是一个数据结构服务 器。

Redis的所有数据都是保存在内存中，然后不定期的通过异步方式保存到磁盘上(这称为“半持久化模式”)；也可以把每一次数据变化都写入到一个append only file(aof)里面(这称为“全持久化模式”)。

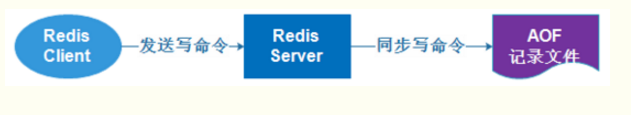
由于Redis的数据都存放在内存中，如果没有配置持久化，redis重启后数据就全丢失了，于是需要开启redis的持久化功能，将数据保存到磁 盘上，当redis重启后，可以从磁盘中恢复数据。redis提供两种方式进行持久化，一种是RDB持久化（原理是将Reids在内存中的数据库记录定时 dump到磁盘上的RDB持久化），另外一种是AOF（append only file）持久化（原理是将Reids的操作日志以追加的方式写入文件）。那么这两种持久化方式有什么区别呢，改如何选择呢？网上看了大多数都是介绍这两 种方式怎么配置，怎么使用，就是没有介绍二者的区别，在什么应用场景下使用。

### 两种持久化的不同

RDB持久化是指在指定的时间间隔内将内存中的数据集快照写入磁盘，实际操作过程是fork一个子进程，先将数据集写入临时文件，写入成功后，再替换之前的文件，用二进制压缩存储。



AOF持久化以日志的形式记录服务器所处理的每一个写、删除操作，查询操作不会记录，以文本的方式记录，可以打开文件看到详细的操作记录。



### 优缺点

RDB存在哪些优势呢？

1). 一旦采用该方式，那么你的整个Redis数据库将只包含一个文件，这对于文件备份而言是非常完美的。比如，你可能打算每个小时归档一次最近24小时的数 据，同时还要每天归档一次最近30天的数据。通过这样的备份策略，一旦系统出现灾难性故障，我们可以非常容易的进行恢复。

2). 对于灾难恢复而言，RDB是非常不错的选择。因为我们可以非常轻松的将一个单独的文件压缩后再转移到其它存储介质上。

3). 性能最大化。对于Redis的服务进程而言，在开始持久化时，它唯一需要做的只是fork出子进程，之后再由子进程完成这些持久化的工作，这样就可以极大的避免服务进程执行IO操作了。

4). 相比于AOF机制，如果数据集很大，RDB的启动效率会更高。

RDB又存在哪些劣势呢？

1). 如果你想保证数据的高可用性，即最大限度的避免数据丢失，那么RDB将不是一个很好的选择。因为系统一旦在定时持久化之前出现宕机现象，此前没有来得及写入磁盘的数据都将丢失。

2). 由于RDB是通过fork子进程来协助完成数据持久化工作的，因此，如果当数据集较大时，可能会导致整个服务器停止服务几百毫秒，甚至是1秒钟。

AOF的优势有哪些呢？

1). 该机制可以带来更高的数据安全性，即数据持久性。Redis中提供了3中同步策略，即每秒同步、每修改同步和不同步。事实上，每秒同步也是异步完成的，其效率也是非常高的，所差的是一旦系统出现宕机现象，那么这一秒钟之内修改的数据将会丢失。而每修改同步，我们可以将其视为同步持久化，即每次发生的数据变化都会被立即记录到磁盘中。可以预见，这种方式在效率上是最低的。至于无同步，无需多言，我想大家都能正确的理解它。

2). 由于该机制对日志文件的写入操作采用的是append模式，因此在写入过程中即使出现宕机现象，也不会破坏日志文件中已经存在的内容。然而如果我们本次操作只是写入了一半数据就出现了系统崩溃问题，不用担心，在Redis下一次启动之前，我们可以通过redis-check-aof工具来帮助我们解决数据一致性的问题。

3). 如果日志过大，Redis可以自动启用rewrite机制。即Redis以append模式不断的将修改数据写入到老的磁盘文件中，同时Redis还会创建一个新的文件用于记录此期间有哪些修改命令被执行。因此在进行rewrite切换时可以更好的保证数据安全性。

4). AOF包含一个格式清晰、易于理解的日志文件用于记录所有的修改操作。事实上，我们也可以通过该文件完成数据的重建。

有哪些呢？

1). 对于相同数量的数据集而言，AOF文件通常要大于RDB文件。RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。

2). 根据同步策略的不同，AOF在运行效率上往往会慢于RDB。总之，每秒同步策略的效率是比较高的，同步禁用策略的效率和RDB一样高效。

二者选择的标准，就是看系统是愿意牺牲一些性能，换取更高的缓存一致性（aof），还是愿意写操作频繁的时候，不启用备份来换取更高的性能，待手动运行save的时候，再做备份（rdb）。rdb这个就更有些 eventually consistent的意思了。

### 如何选择RDB和AOF

1、同时开启

Redis先加载aof文件来恢复原始数据,因为AOF数据比RDB更完整,但是AOF存在潜在的bug,如把错误的操作记录写入了aof, 会导出数据恢复失败,所以可以把RDB作为后备数据。

为了考虑性能,可以只在Slave上开启RDB,并且15min备份一次,如果为了避免AOF rewite的10以及阻塞,可以在Redis集群中不开启AOE,靠集群的备份机制来保证可用性,在启动时选取较新的RDB文件,如果集群全部崩溃,会丢失15min前的数据。

2、混合模式

Redis4.0开始支持该模式。

解决的问题: Redis在重启时通常是加载AOF文件,但加载速度慢。因为RDB数据不完整,所以加载AOF.

开启方式: aof-use-rdb-preamble true

开启后, AOF在重写时会直接读取RDB中的内容。

运行过程:通过bgrwriteaof完成,不同的是当开启混合持久化后

1,子进程会把内存中的数据以RDB的方式写入aof中

2,把重写缓冲区中的增量命令以AOF方式写入到文件

3·将含有RDB个数和AOF格数的AOF数据覆盖旧的AOE文件

新的AOF文件中,一部分数据来自RDB文件,一部分来自Redis运行过程时的增量数据

**数据恢复**

当我们开启了混合持久化时,启动Redis依然优先加载2文件, aof文件加载可能有两种情况如下:

aof文件开头是rb的格式,先加载rdb内容再加载剩余的aof

aof文件开头不是rdb的格式,直接以aof格式加载整个文件’

优点:既能快速备份又能避免大量数据丢失

缺点: RDB是压缩格式, AOF在读取它时可读性较差

### 二者如何动态切换

在Redis 2.2或以上版本,可以在不重启的情况下,从RDB切换到AOE:

为最新的dump.rdh文件创建一个备份，将备份放到一个安全的地方

CP dump.rdb dump.rdb

执行下面两条命令

#开启aof 1

redis-cli config set appendonly yes

#关闭rdb

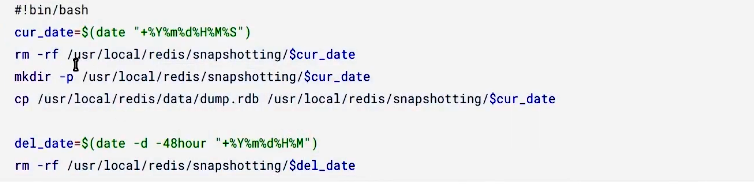
redis-cli config set save “”

确保写命令会被正确地追加到AOE文件的末尾

执行的第一条命令开启了AOE功能: Redis会阻塞直到初始AOE文件创建完成为止,之后Redis会继续处理命令请求,并开始将写入命令追加到AOE文件末尾

### Redis 容灾备份

1、rdb日志备份，编写脚本定时备份





### 优化方案

1、独立部署问题，硬盘优化，减少其他进程争夺内存资源。

2、缓存禁用持久化。根据情况下决定是否禁用。

3、主从模式,从节点持久化。主节点关闭aof。

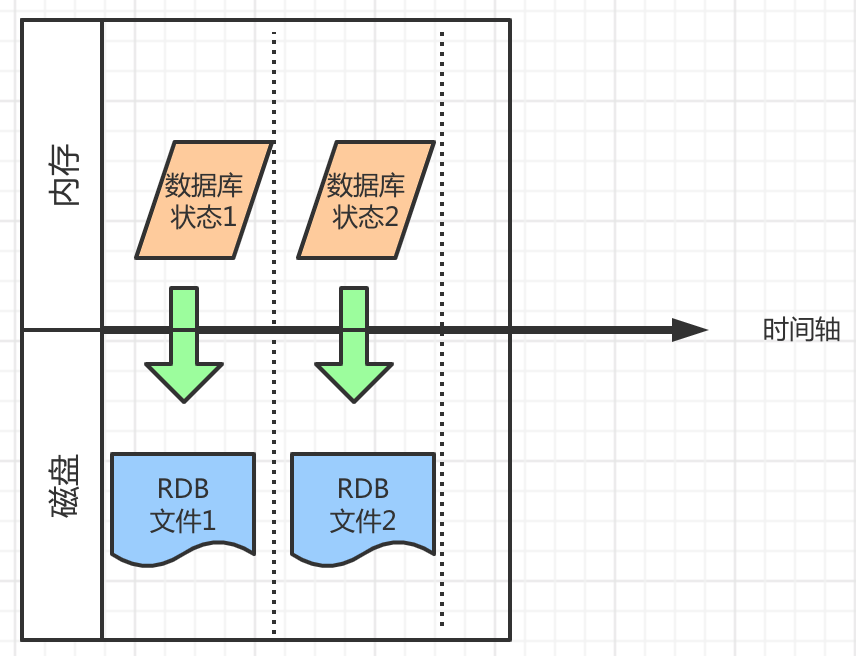
4、优化fork处理，降低rewrite 的评率。

### 持久化的问题

https://zhuanlan.zhihu.com/p/333836460

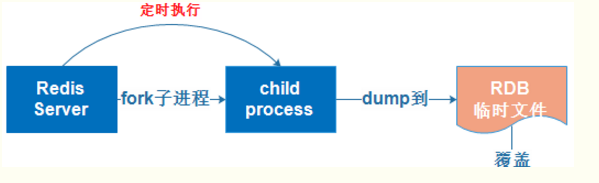
## RDB(Redis Database)

### RDB



RDB是Redis Database 的缩写，其作用是在某一个时间点，将Redis存储在内存中的数据生成快照并存储到磁盘等介质上，存在这个磁盘介质上的文件就是RDB文件。“快照”顾名思义就是好像照相一样保存当时的数据，这里的RDB文件是一个二进制的文件，并且是经过压缩的。﻿因为RDB文件是保存在硬盘中的，即使Redis服务器进程退出，甚至运行Redis服务器的计算机宕机，但只要RDB文件仍然存在，Redis服务器就可以用它来还原数据库状态。如图1 所示，可以想象Redis数据库在时间轴上有位于不同时间点的时候都有一个数据库状态，可以把它们想象成一个个切片。图上标注出两个时间点的两个数据库切片，RDB持久化做的事情就是顺着绿色箭头的方向将数据库状态的“切片”以RDB文件的形式保存到磁盘中。

RDB持久化是指在指定的时间间隔内将内存中的数据集快照写入磁盘，实际操作过程是fork一个子进程，先将数据集写入临时文件，写入成功后，再替换之前的文件，用二进制压缩存储。



### RDB文件结构

虽然说RDB是一个压缩过的二进制文件，但是它的文件结构也需要有基本的了解，这样有助于我们理解其发挥的作用。如表格1所示，RDB文件由5个部分组成，按照从左到右的顺序依次是：

文件最开头是“REDIS”部分，其长度为5个字节，保存着“REDIS”五个字符。通过这五个字符，程序可以在载入文件时可以判断所载入的文件是否是RDB文件。

接下来是“db\_version”长度为4字节，是一个字符串表示的整数，它记录了RDB文件的版本号，例如："0006"就代表RDB文件的版本为第六版。

“databases”中可以包含着零个或任意多个数据库。

“EOF”常量的长度为1字节，是 RDB文件正文结束的标识，当载入程序读取到个值的时，就意味着数据库的所有键值对都已经加载完毕了。

“check\_sum”是一个8字节长的无符号整数，保存着一个校验和。这个校验和是通过对“REDIS”、“db\_version”、“databases”、以及“EOF”四个部分的内容进行计算得出的。Redis服务器在载入RDB文件时，会将载入数据所计算出的校验和与check\_sum所记录的校验和进行对比，以此来判断RDB文件是否损坏。

文件结构



﻿如表格2所示，其表示一个databases部分为空的RDB文件：文件以"REDIS"开头，表示这是一个RDB文件，之后的"0006"表示数据库版本是第六版。因为databases为空所以这里没有数据库的信息，所以版本号之后直接跟着“EOF”常量，最后的6265312314761934562是文件的校验和。



### RDB配置

Redis会将数据集的快照dump到**dump.rdb**文件中。此外，我们也可以通过配置文件来修改Redis服务器dump快照的频率，在打开redis.conf文件之后，我们搜索save，可以看到下面的配置信息：

**配置信息**

|  |
| --- |
| dbfilename dump.rdb 镜像备份文件的文件名。  dir /usr/local/redis/data 数据库镜像备份的文件放置的路径  #这里的save可以有多个，满足一个条件就执行。  save 900 1 #在900秒(15分钟)之后，如果至少有1个key发生变化，则dump内存快照。  save 300 10 #在300秒(5分钟)之后，如果至少有10个key发生变化，则dump内存快照。  save 60 10000 #在60秒(1分钟)之后，如果至少有10000个key发生变化，则dump内存快照。 |

生产环境配置:

### 触发RDB

**Bgsave:该命令是非阻塞的，fork一个子进程执行做dump。**

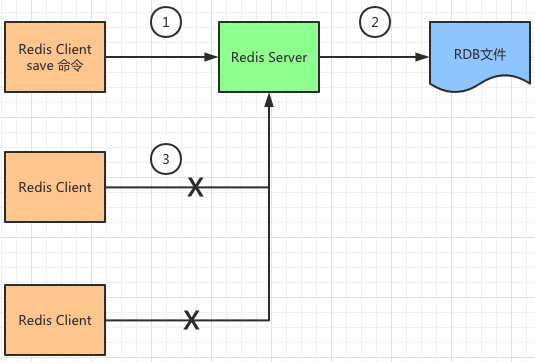
**save配置:该命令是阻塞的,执行命令的时候不能做其他操作。**

**配置文件：根据配置文件去执行。**

**Shutdown:客户端发送shutdown,如果没有开启 AOF 持久化功能则自动执行 bgsave。**

**当有主从架构时,从服务器向主服务器发送sync命令来执行复制操作时,主服务器会执行bgsave操作**

**save同步方式触发RDB持久化**

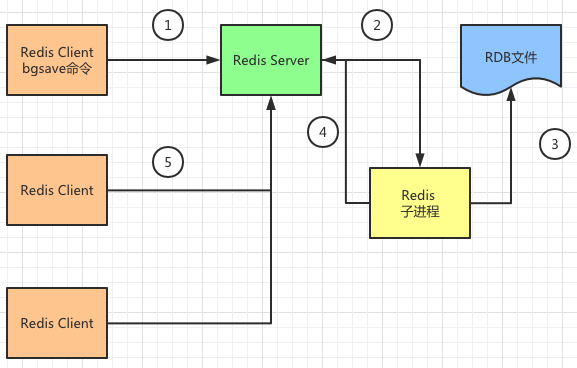


1、Redis Client端通过向Redis Server 发起save命令请求RDB持久化操作。

2、Redis Server接受到命令以后，将当前数据库快照保存到RDB文件中。

3、由于save命令是同步操作，因此如果此时有其他Redis Client也向Redis Server发起save操作，会被阻塞直到第一个Redis Client完成save命令为止。

bgsave同步方式触发RDB持久化



1、依旧是Redis Client发起命令，不过命令改成了bgsave（background save有后台运行的意思），照旧请求Redis Server。

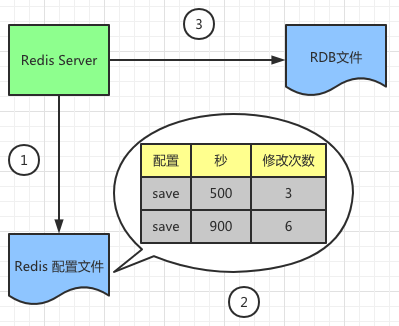
2、Redis Server接受到请求以后会fork出一个Redis子进程。

3、这个子进程用来创建RDB文件。由于这个过程是异步的，因此Redis Server在启动子进程以后还可以接受其他请求。

4、Redis 子进程创建RDB文件以后会把成功的消息返回给Redis Server。

5、由于bgsave命令是异步操作，如果此时有其他Redis Client同时请求Redis Server并不会被阻塞。Redis Server会响应请求，同样也会fork出对应的子进程进行RDB文件的创建。

**自动配置化的方式触发：**

****

这种方式可以理解为读取配置文件的方式。看下面的三个步骤：

1、Redis Server直接读取Redis 配置文件的内容，获取RDB持久化的信息。

2、在Redis配置文件中配置了对应的save命令用来代替Redis Client请求的命令。其配置内容包括save命令、秒和修改次数。按照图中的例子来说，“save 500 3” 的意思是，在500秒的时间内如果Redis 数据库有3次修改就进行save请求，也就是请求RDB的持久化操作。

3、一旦满足配置文件中的条件，Redis Server就会执行对应的save操作进行持久化

一般而言Redis的配置信息会放到redis.conf配置文件中进行存储，其包含很多内容，这里我们就RDB持久化的部分给大家做简单介绍。redis.conf文件中找到“SNAPSHOTTING”（快照）的部分。看如下几个配置项：

|  |
| --- |
| # 900秒内有1次修改、300秒内有10次修改、60秒内有10000次修改  #满足任何以上条件，触发RDB持久化。  save 900 1  save 300 10  save 60 10000  # 当快照操作bgsave出错时，是否停止持久化？yes 表示“是”，no表示“否”。  stop-writes-on-bgsave-error yes  # 是否压缩？yes表示“是”，no表示“否”，默认选择yes。  rdbcompression yes  # 对rdb数据进行校验, 表示写入文件和读取文件时是否开启 RDB 文件检查。  # yes表示“是”，no表示“否”，默认选择yes。  # 选择yes表示在Redis加载RDB需要检查文件是否损坏，如果存在损坏会停止启动。  rdbchecksum yes  # 设置rdb的文件名  dbfilename dump.rdb  # RDB文件的路径，如果不单独指定，默认是redis的启动路径。  dir ./ |

关于RDB持久化恢复Redis数据方面也比较简单，将RDB持久化文件 (例如：dump.rdb) 移动到 Redis 安装目录并启动Redis服务就可以了。可以通过 Redis 中的“CONFIG GET dir”命令获取Redis的安装目录。

由于RDB是一个压缩的二进制文件，其代表Redis在某一个时间点上的快照。其适合数据库备份和全量复制的场景。比如定期给数据库进行备份，把RDB文件拷贝到其他的服务器上，以及用于灾备。同样是因为压缩的原因，RDB的加载速度比AOF也要快。

### RDB的优缺点

优点:

1、RDB存储的方式为二进制文件

2、fork子进程性能最大化。

3、启动效率高。

缺点:

1、生成快照的时机问题。

2、fork子进程的开销问题。

## AOF(append only file)

### AOF

上面介绍了RDB的执行方式和流程，这种方式没有办法做到实时持久化的要求。因为无论是save还是bgsave每次运行都要消耗大量的资源（CPU、内存、磁盘）。随着数据库本身容量的增加每次备份的数据量也随之增加。同时RDB是二进制保存，当Redis版本演进过程中有多个格式的RDB版本，会存在老版本RDB与新版本格式兼容的问题。正式因为RDB的这些问题，Redis提出了AOF的持久化方式。AOF（append only file），是以日志的方式记录每次写入的命令，在Redis Server启动的时候会重新执行AOF文件中的命令，从而达到恢复数据的目的。AOF可以解决数据持久化的实时性问题，也是当前Redis主流的持久化方式。

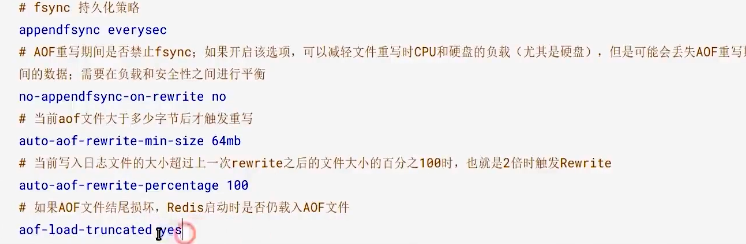
### AOF配置

AOF配置默认关闭。

|  |
| --- |
| #aop持久化开启  appendonly yes  #aop持久化文件名称  appendfilename "appendonly.aof"  #数据库镜像备份的文件放置的路径。  dir /usr/local/redis/data  aof配置重写 |

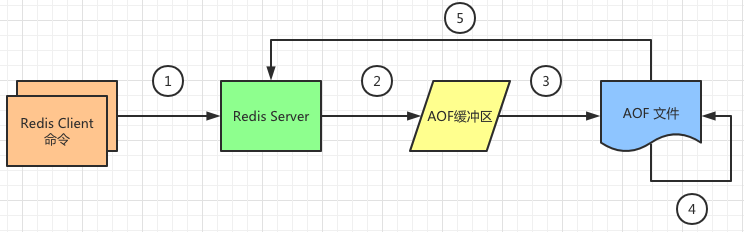
RDB与AOF同时开启 默认先加载AOF的配置文件

A配置重写。



生产环境:一般会打开AOF.

### 持久化流程



Redis Client作为命令的来源，会有多个源头以及源源不断的请求命令。

1、在这些命令到达Redis Server 以后，并不是直接写入AOF文件，会将其这些命令先放入AOF缓存中进行保存。这里的AOF缓冲区实际上是内存中的一片区域，存在的目的是当这些命令达到一定量以后再写入磁盘，避免频繁的磁盘IO操作。

2、AOF缓冲会根据对应的策略将命令写入磁盘上的AOF文件。

3、AOF文件随着写入文件内容的增加，会根据规则进行命令的合并，这里叫做AOF重写，从而起到AOF文件压缩的目的。

4、当Redis Server 服务器重启的时候会从AOF文件载入数据。

### AOF缓冲区同步文件策略

上面提到了Redis 会将命令先写入到AOF缓冲区，再写入AOF文件。这里介绍一下AOF缓冲区同步文件的三个策略。

always 策略：命令写入AOF缓冲区以后会调用系统fsync 操作同步到AOF文件，fsync完成后线程返回。这里的fsync是针对单个文件的操作，在进行磁盘同步的时候会阻塞直到写入磁盘完成以后返回，从而保证数据持久化的完成。Redis-Server每写入一条，就持久化到AOF文件中。

everysec 策略：命令写入AOF缓冲区以后调用write操作，write完成后线程返回。此操作会有专门线程执行每秒执行一次。这里的write操作会触发延迟写（delayed write）机制，Linux 内核提供页缓冲区来提高硬盘IO性能。也就是说write 操作写入系统缓冲区以后就返回了，同步硬盘依赖于操作系统调度机制完成。（Redis默认配置）

no策略：此种刷新策略是根据操作系统来决定的，也就是由操作系统来决定什么时候将缓冲区的数据写入到磁盘中。由于是操作系统来决定持久化，所以这种方式是不可控的。

### AOF 重写

AOF缓冲区会将Redis Client请求的命令源源不断地同步到AOF文件中，同时AOF文件会不断增大，这里就需要AOF重写。AOF重写就是把Redis进程内的数据转化为写命令同步到新的AOF文件的过程。其目的就是使重写后的AOF文件变得更小：

进程内已经超时的数据不会再写入AOF文件中。

旧AOF文件含有的无效命令，可以通过进程内的数据直接生成，新的AOF文件只保留最终的数据写入命令。例如就文件中存在三条命令，它们依次是“set hello A”、 “set hello B”和“set hello C”，对同一个key 进行负值只有最后一句“set hello C”是起效的，所以这三条命令会被“set hello C”一条命令替换，并且保存到新的AOF文件中。

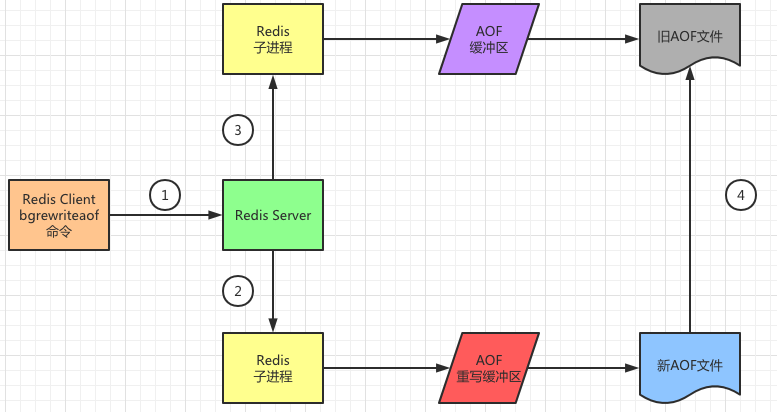
另外，多条写命令可以合并成一个。例如依次存在三个命令：“lpush list A”、 “lpush list B”和“lpush list C”，这里就可以合并为一条命令“lpush list A B C”。

AOF重写不仅降低了文件的占用空间，同时更小的AOF也可以更快地被Redis加载。

说完了AOF重写的定义以后，下面来看看AOF重写的流程。一般而言有两种方式可以执行重写操作，

分别是:bgrewriteaof 命令和AOF重写配置。

### bgrewriteaof命令重写



Redis Client发起bgrewriteaof命令，这个命令是一个异步命令。由于Redis Server 在接受bgrewriteaof命令的同时，还可以接受其他Redis Client的命令，因此后面的第二步和第三步实际上是并行进行的。第二步：进行AOF重写，在同时第三步：还会接受其他非重写的命令请求。

Redis Server接受到这个命令以后，会启动一个Redis的子进程用来执行AOF重写操作。这个重写过程实际上是针对Redis内存中的数据进行回溯，也就是下方红色区域的“AOF重写缓冲区”

正如在第一步中提到的，在进行第二步的同时第三步还在接受客户端的请求并且通过“AOF缓冲区”保存到“旧AOF文件”中。

最终，完成AOF重写操作以后将“新AOF文件”写入到“旧AOF文件”中完成AOF重写。

## 企业级的持久化

### 数据的恢复

注:如果通过RDB恢复，一定要停掉AOF配置。

1如果是redis进程挂掉,那么重启redis进程即可,直接基于AOF日志文件恢复数据,

2)如果是redis进程所在机器挂掉,那么重启机器后,尝试重启redis进程,尝试直接基于AOF日志文件进行数据恢复

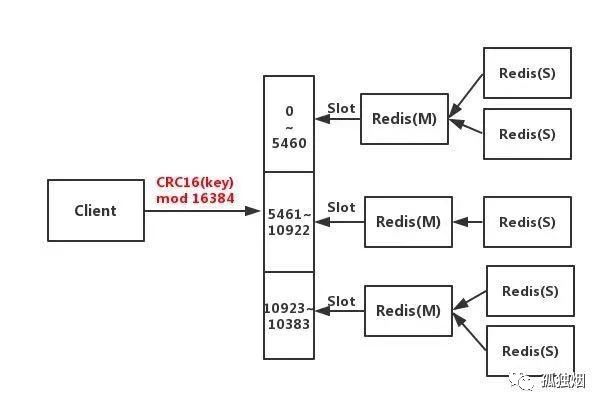
3) 如果redis当前最新的AOF和RDB文件出现了丢失/损坏,那么可以尝试基于该机器上当前的某个最新的RDB数据副本进行数据恢复.

4、如果当前机器上的所有RDB文件全部损坏,那么从远程的云服务上拉取最新的RDB快照回来恢复数据

5、如果是发现有重大的数据错误,比如某个小时上线的程序一下子将数据全部污染了,数据全错了,那么可以选择某个更早的时间点,对数据进行恢复。

步骤:停止redis,关闭aof,拷贝rdb备份,重启redis,确认数据恢复,直接在命令行热修改redis配置,打开aof。

## 为什么是16384个槽



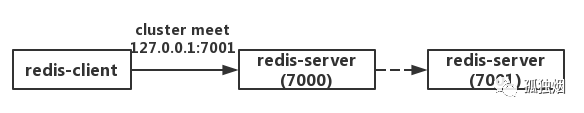
对于客户端请求的key，根据公式HASH\_SLOT=CRC16(key) mod 16384，计算出映射到哪个分片上，然后Redis会去相应的节点进行操作！

ps:CRC16算法产生的hash值有16bit，该算法可以产生2^16-=65536个值。换句话说，值是分布在0~65535之间。那作者在做mod运算的时候，为什么不mod65536，而选择mod16384？

我们回忆一下Redis Cluster的工作原理！

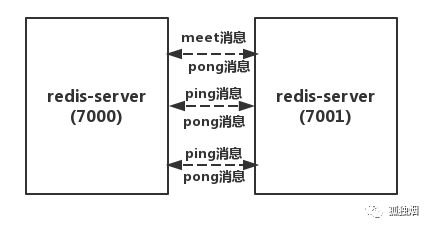
这里要先将节点握手讲清楚。我们让两个redis节点之间进行通信的时候，需要在客户端执行下面一个命令

127.0.0.1:7000>cluster meet 127.0.0.1:7001



意思很简单，让7000节点和7001节点知道彼此存在！

在握手成功后，连个节点之间会**定期**发送ping/pong消息，交换**数据信息**，如下图所示。



在这里，我们需要关注三个重点。

(1)交换什么数据信息

(2)数据信息究竟多大

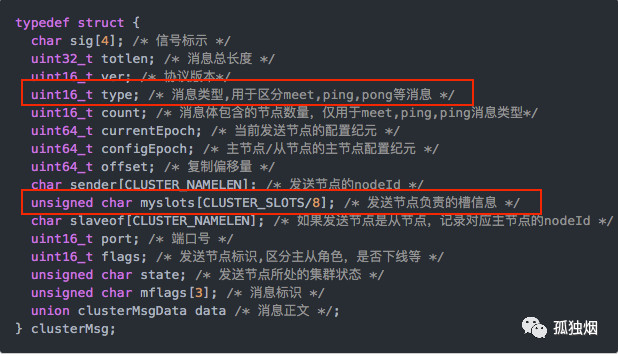
(3)定期的频率什么样

到底在交换什么数据信息？

交换的数据信息，由消息体和消息头组成。

消息体无外乎是一些节点标识啊，IP啊，端口号啊，发送时间啊。这与本文关系不是太大，我不细说。

我们来看消息头，结构如下



注意看红框的内容，type表示消息类型。

另外，消息头里面有个myslots的char数组，长度为16383/8，这其实是一个bitmap,每一个位代表一个槽，如果该位为1，表示这个槽是属于这个节点的。

到底数据信息究竟多大？

在消息头中，最占空间的是myslots[CLUSTER\_SLOTS/8]。这块的大小是:

16384÷8÷1024=2kb

那在消息体中，会携带一定数量的其他节点信息用于交换。  
那这个其他节点的信息，到底是几个节点的信息呢？  
约为集群总节点数量的1/10，至少携带3个节点的信息。  
这里的重点是:节点数量越多，消息体内容越大。

消息体大小是10个节点的状态信息约1kb。

那定期的频率是什么样的？

redis集群内节点，每秒都在发ping消息。规律如下

(1)每秒会随机选取5个节点，找出最久没有通信的节点发送ping消息

(2)每100毫秒(1秒10次)都会扫描本地节点列表，如果发现节点最近一次接受pong消息的时间大于cluster-node-timeout/2 则立刻发送ping消息

因此，每秒单节点发出ping消息数量为

数量=1+10\*num（node.pong\_received>cluster\_node\_timeout/2）

那大致带宽损耗如下所示，图片来自《Redis运维与实现》



(1)如果槽位为65536，发送心跳信息的消息头达8k，发送的心跳包过于庞大。如上所述，在消息头中，最占空间的是myslots[CLUSTER\_SLOTS/8]。当槽位为65536时，这块的大小是: 65536÷8÷1024=8kb

因为每秒钟，redis节点需要发送一定数量的ping消息作为心跳包，如果槽位为65536，这个ping消息的消息头太大了，浪费带宽。

(2)redis的集群主节点数量基本不可能超过1000个。

如上所述，集群节点越多，心跳包的消息体内携带的数据越多。如果节点过1000个，也会导致网络拥堵。因此redis作者，不建议redis cluster节点数量超过1000个。

那么，对于节点数在1000以内的redis cluster集群，16384个槽位够用了。没有必要拓展到65536个。

(3)槽位越小，节点少的情况下，压缩比高

Redis主节点的配置信息中，它所负责的哈希槽是通过一张bitmap的形式来保存的，在传输过程中，会对bitmap进行压缩，但是如果bitmap的填充率slots / N很高的话(N表示节点数)，bitmap的压缩率就很低。

如果节点数很少，而哈希槽数量很多的话，bitmap的压缩率就很低。

## 性能测试

### Redis-benchmark(性能测试工具)



### 单机测试

./redis-benchmark -a 123456 -h 192.168.205.128 -p 6379 -t set,get -r 1000000 -n 1000000 -c 1000

|  |
| --- |
| ====== SET ======  //请求数量在9.47完成  1000000 requests completed in 9.47 seconds  //1000并发  1000 parallel clients  //操作的字节  3 bytes payload  //长连接  keep alive: 1  //rdb开启  host configuration "save":  //aof开启  host configuration "appendonly": yes  multi-thread: no  99.97% <= 25 milliseconds  99.98% <= 26 milliseconds  99.99% <= 27 milliseconds  99.99% <= 28 milliseconds  100.00% <= 29 milliseconds  100.00% <= 29 milliseconds  //每秒处理的请求。  105574.32 requests per second  ====== GET ======  1000000 requests completed in 9.58 seconds  1000 parallel clients  3 bytes payload  keep alive: 1  host configuration "save":  host configuration "appendonly": yes  multi-thread: no  99.95% <= 23 milliseconds  99.97% <= 24 milliseconds  99.99% <= 25 milliseconds  100.00% <= 26 milliseconds  100.00% <= 26 milliseconds  104362.35 requests per second |

### 集群测试

./redis-benchmark -a 123456 -h 192.168.205.128 -p 6373 -t set,get -r 1000000 -n 1000000 -c 1000

|  |
| --- |
| ====== SET ======  1000000 requests completed in 10.87 seconds  1000 parallel clients  3 bytes payload  keep alive: 1  host configuration "save":  host configuration "appendonly": yes  multi-thread: no  99.97% <= 28 milliseconds  99.98% <= 29 milliseconds  99.98% <= 30 milliseconds  99.99% <= 31 milliseconds  100.00% <= 32 milliseconds  100.00% <= 32 milliseconds  92021.72 requests per second  ====== GET ======  1000000 requests completed in 10.27 seconds  1000 parallel clients  3 bytes payload  keep alive: 1  host configuration "save":  host configuration "appendonly": yes  multi-thread: no  99.96% <= 32 milliseconds  99.97% <= 33 milliseconds  99.98% <= 34 milliseconds  99.98% <= 35 milliseconds  99.99% <= 36 milliseconds  100.00% <= 36 milliseconds  97333.07 requests per second |

### 对比

Set :

单机9.47

集群10.87

Get :

单机9.58

集群:10.27

## Redis的分布式锁实现

Redis分布式锁的基本流程并不难理解，但要想写得尽善尽美，也并不是那么容易。在这里，我们需要先了解分布式锁实现的三个核心要素：

### 加锁

最简单的方法是使用setnx命令。key是锁的唯一标识，按业务来决定命名。比如想要给一种商品的秒杀活动加锁，可以给key命名为 “lock\_sale\_商品ID” 。而value设置成什么呢？我们可以姑且设置成1。加锁的伪代码如下：

setnx（key，1）

当一个线程执行setnx返回1，说明key原本不存在，该线程成功得到了锁；当一个线程执行setnx返回0，说明key已经存在，该线程抢锁失败。

### 解锁

有加锁就得有解锁。当得到锁的线程执行完任务，需要释放锁，以便其他线程可以进入。释放锁的最简单方式是执行del指令，伪代码如下：

del（key）

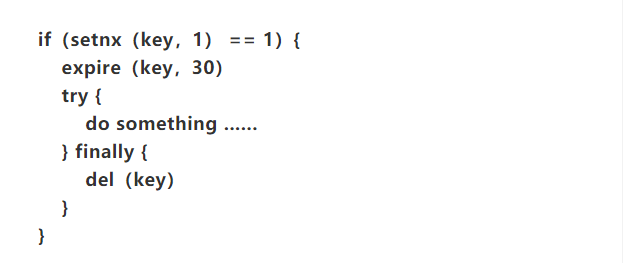
释放锁之后，其他线程就可以继续执行setnx命令来获得锁。

### 锁超时

锁超时是什么意思呢？如果一个得到锁的线程在执行任务的过程中挂掉，来不及显式地释放锁，这块资源将会永远被锁住，别的线程再也别想进来。

所以，setnx的key必须设置一个超时时间，以保证即使没有被显式释放，这把锁也要在一定时间后自动释放。setnx不支持超时参数，所以需要额外的指令，伪代码如下：

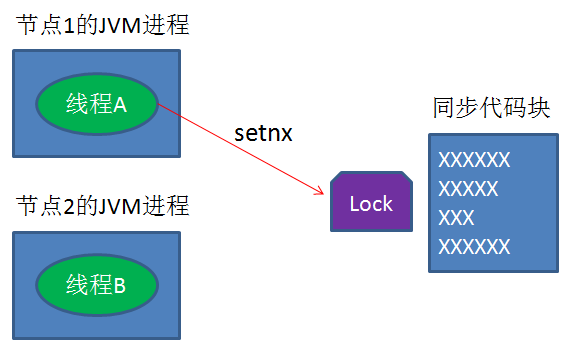
expire（key， 30）



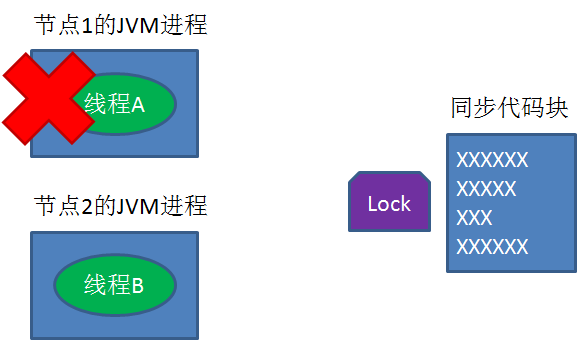
### 问题

1. setnx和expire的非原子性

设想一个极端场景，当某线程执行setnx，成功得到了锁：



setnx刚执行成功，还未来得及执行expire指令，节点1 Duang的一声挂掉了。



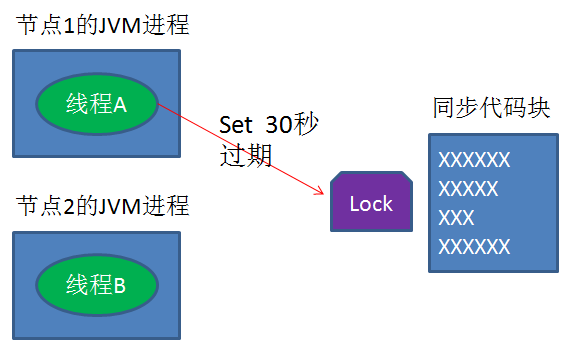
这样一来，这把锁就没有设置过期时间，变得“长生不老”，别的线程再也无法获得锁了。

怎么解决呢？setnx指令本身是不支持传入超时时间的，幸好Redis 2.6.12以上版本为set指令增加了可选参数，伪代码如下：

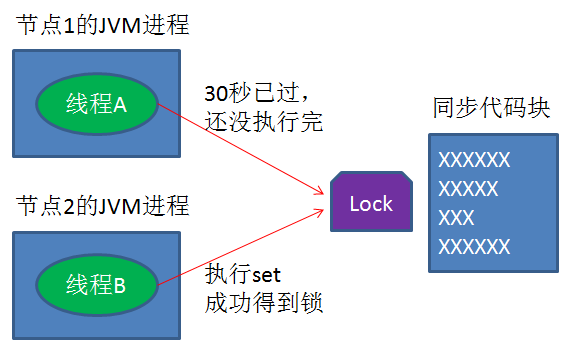
set（key，1，30，NX）

**del 导致误删**

又是一个极端场景，假如某线程成功得到了锁，并且设置的超时时间是30秒。



如果某些原因导致线程A执行的很慢很慢，过了30秒都没执行完，这时候锁过期自动释放，线程B得到了锁。



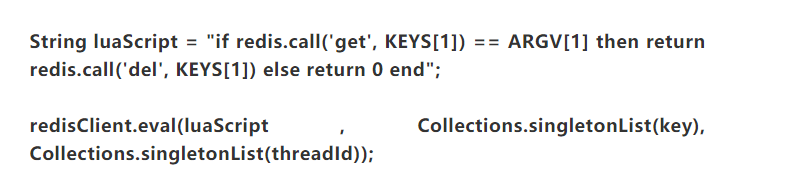
随后，线程A执行完了任务，线程A接着执行del指令来释放锁。但这时候线程B还没执行完，线程A实际上删除的是线程B加的锁。

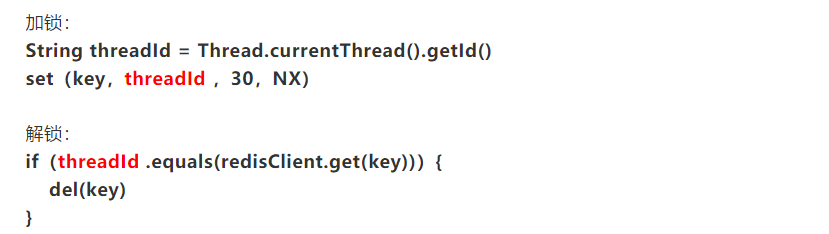
怎么避免这种情况呢？可以在del释放锁之前做一个判断，验证当前的锁是不是自己加的锁。

至于具体的实现，可以在加锁的时候把当前的线程ID当做value，并在删除之前验证key对应的value是不是自己线程的ID。

但是，这样做又隐含了一个新的问题，判断和释放锁是两个独立操作，不是原子性。

我们都是追求极致的程序员，所以这一块要用Lua脚本来实现：





**这样一来，验证和删除过程就是原子操作了。**

## 缓存和数据库一致性实战

<https://www.jianshu.com/p/93f16ca6b396>

## redis问题

### 主从数据一致性

主从网络延时:

主多从少:情况下，进行部分重同步(增量同步)

主少从多情况下,进行全量复制

### 主从数据延迟

1、编写外部程序监听主从节点的复制偏移量,延迟较大时发出报警或通知客户端，切换到主节点或其他节点

2、设置从节点slave-serve-stale-data为no,除INFO和SLAVOF命令之外的任何请求都会返回一个错误progress with master in progress .

### 脏数据

1、Redis删除机制导致(惰性、定时、主动删除等)

2、从节点写入。

1、忽略，查询的时候不去做任何操作，允许有一定错误，例如12306查询余票。

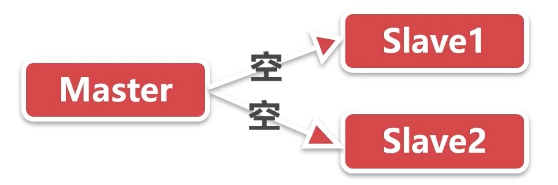
2、如果业务一定会操作数据，选择性强制读主，从节点间接变为了备份服务器。

3、从节点只读，规避从节点写入脏数据

目前Redis读取数据之前检查键过期时间来决定是否返回数据

### 数据的安全性

关闭主节点持久化会提升性能,同时会带来复制的安全性问题



主节点关闭持久化,主节点挂掉之后，会从salve做复制请求，把空的数据度过来，master变为空。

解决方案：主节点不自动重启，比如Docker或者脚本等

### 性能优化

规避全量复制：

1、第一次全量复制解决方案:低峰时段挂载Slave。

2、选举Slave为主节点

3、增大复制缓冲区

规避复制风暴：从节点同一时间向主节点做请求复制。

单主节点复制风暴：主节点重启，多从节点全量复制

解决方法:选举Slave为主节点、树状复制结构

单机多主复制风暴:一台机器,多个主节点(树状复制架构)

解决方案：把主节点分散在多台机器上

## Redis生产环境

### 生产环境启动

将redis作为一个系统的daemon进程去运行,每次系统启动,redis进程一起启动

1、redis utils目录下，有个redis\_init\_script脚本

2、将redis\_init\_script脚本拷贝到linux的/etc/init.d目录中，将redis\_init\_script重命名为redis\_6379(命令是mv redis\_init\_script redis\_6379)，6379是我们希望这个redis实例监听的端口号

3、修改redis\_6379脚本的第6行的REDISPORT，设置为相同的端口号（默认就是6379）

4、创建两个目录：/etc/redis（存放redis的配置文件），/var/redis/6379（存放redis的持久化文件）

5、修改redis配置文件（默认在根目录下，redis.conf），拷贝到/etc/redis目录中，修改名称为6379.conf

6、修改6379.conf的部分配置为生产环境

daemonize yes 让redis以daemon进程运行

pidfile /var/run/redis\_6379.pid 设置redis的pid文件位置

port 6379 设置redis的监听端口号

dir /var/redis/6379 设置持久化文件的存储位置

7、启动redis，执行cd /etc/init.d, chmod 777 redis\_6379，./redis\_6379 start

8、确认redis进程是否启动，ps -ef | grep redis

9、让redis跟随系统启动自动启动，在目录/etc/init.d下执行, 设置为开机自启

sudo update-rc.d redis\_6379 e

ubunto:

|  |
| --- |
| ### BEGIN INIT INFO  # Provides: redis\_6379  # Required-Start: $network $remote\_fs $syslog $time  # Required-Stop:  # Default-Start: 2 3 4 5  # Default-Stop: 0 1 6  # Short-Description: Redis data structure server  # Description: Redis data structure server. See https://redis.io  ### END INIT INFO |

# 如何保证缓存和数据库一致性

### 不更新缓存，而是删除缓存

大部分观点认为，做缓存不应该是去更新缓存，而是应该删除缓存，然后由下个请求去去缓存，发现不存在后再读取数据库，写入缓存。

**原因一：线程安全角度**

**同时有请求A和请求B进行更新操作，那么会出现**

（1）线程A更新了数据库

（2）线程B更新了数据库

（3）线程B更新了缓存

（4）线程A更新了缓存

这就出现请求A更新缓存应该比请求B更新缓存早才对，但是因为网络等原因，B却比A更早更新了缓存。这就导致了脏数据，因此不考虑。

**原因二：业务场景角度**

（1）如果你是一个写数据库场景比较多，而读数据场景比较少的业务需求，采用这种方案就会导致，数据压根还没读到，缓存就被频繁的更新，浪费性能。

（2）如果你写入数据库的值，并不是直接写入缓存的，而是要经过一系列复杂的计算再写入缓存。那么，每次写入数据库后，都再次计算写入缓存的值，无疑是浪费性能的。显然，删除缓存更为适合。

其实如果业务非常简单，只是去数据库拿一个值，写入缓存，那么更新缓存也是可以的。但是，淘汰缓存操作简单，并且带来的副作用只是增加了一次cache miss，建议作为通用的处理方式。

### 先操作缓存，还是先操作数据库

那么问题就来了，我们是先删除缓存，然后再更新数据库，还是先更新数据库，再删缓存呢？

对于一个不能保证事务性的操作，一定涉及“哪个任务先做，哪个任务后做”的问题，解决这个问题的方向是：如果出现不一致，谁先做对业务的影响较小，就谁先执行。

假设先淘汰缓存，再写数据库：第一步淘汰缓存成功，第二步写数据库失败，则只会引发一次Cache miss。

假设先写数据库，再淘汰缓存：第一步写数据库操作成功，第二步淘汰缓存失败，则会出现DB中是新数据，Cache中是旧数据，数据不一致

。

沈剑老师说的没有问题，不过没完全考虑好并发请求时的数据脏读问题，让我们再来看看孤独烟老师《分布式之数据库和缓存双写一致性方案解析》：

**先删缓存，再更新数据库**

该方案会导致请求数据不一致

同时有一个请求A进行更新操作，另一个请求B进行查询操作。那么会出现如下情形:

（1）请求A进行写操作，删除缓存

（2）请求B查询发现缓存不存在

（3）请求B去数据库查询得到旧值

（4）请求B将旧值写入缓存

（5）请求A将新值写入数据库

上述情况就会导致不一致的情形出现。而且，如果不采用给缓存设置过期时间策略，该数据永远都是脏数据。

所以先删缓存，再更新数据库并不是一劳永逸的解决方案，再看看先更新数据库，再删缓存这种方案怎么样？

先更新数据库，再删缓存这种情况不存在并发问题么？

不是的。假设这会有两个请求，一个请求A做查询操作，一个请求B做更新操作，那么会有如下情形产生

（1）缓存刚好失效

（2）请求A查询数据库，得一个旧值

（3）请求B将新值写入数据库

（4）请求B删除缓存

（5）请求A将查到的旧值写入缓存

ok，如果发生上述情况，确实是会发生脏数据。

发生上述情况有一个先天性条件，就是步骤（3）的写数据库操作比步骤（2）的读数据库操作耗时更短，才有可能使得步骤（4）先于步骤（5）。可是，大家想想，数据库的读操作的速度远快于写操作的（不然做读写分离干嘛，做读写分离的意义就是因为读操作比较快，耗资源少），因此步骤（3）耗时比步骤（2）更短，这一情形很难出现。

**先更新数据库，再删缓存依然会有问题，不过，问题出现的可能性会因为上面说的原因，变得比较低！**

所以，如果你想实现基础的缓存数据库双写一致的逻辑，那么在大多数情况下，在不想做过多设计，增加太大工作量的情况下，请先更新数据库，再删缓存!

### 非要数据库和缓存数据强一致怎么办

那么，如果我非要保证绝对一致性怎么办，先给出结论：

没有办法做到绝对的一致性，这是由CAP理论决定的，缓存系统适用的场景就是非强一致性的场景，所以它属于CAP中的AP。

所以，我们得委曲求全，可以去做到BASE理论中说的最终一致性。

最终一致性强调的是系统中所有的数据副本，在经过一段时间的同步后，最终能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性

大佬们给出了到达最终一致性的解决思路，主要是针对上面两种双写策略（先删缓存，再更新数据库/先更新数据库，再删缓存）导致的脏数据问题，进行相应的处理，来保证最终一致性。

### 缓存延时双删

问：先删除缓存，再更新数据库中避免脏数据？

答案：采用延时双删策略。

上文我们提到，在先删除缓存，再更新数据库的情况下，如果不采用给缓存设置过期时间策略，该数据永远都是脏数据。

那么延时双删怎么解决这个问题呢？

（1）先淘汰缓存

（2）再写数据库（这两步和原来一样）

（3）休眠1秒，再次淘汰缓存

这么做，可以将1秒内所造成的缓存脏数据，再次删除。

那么，这个1秒怎么确定的，具体该休眠多久呢？

针对上面的情形，读者应该自行评估自己的项目的读数据业务逻辑的耗时。然后写数据的休眠时间则在读数据业务逻辑的耗时基础上，加几百ms即可。这么做的目的，就是确保读请求结束，写请求可以删除读请求造成的缓存脏数据。

**如果你用了mysql的读写分离架构怎么办？**

ok，在这种情况下，造成数据不一致的原因如下，还是两个请求，一个请求A进行更新操作，另一个请求B进行查询操作。

（1）请求A进行写操作，删除缓存

（2）请求A将数据写入数据库了，

（3）请求B查询缓存发现，缓存没有值

（4）请求B去从库查询，这时，还没有完成主从同步，因此查询到的是旧值

（5）请求B将旧值写入缓存

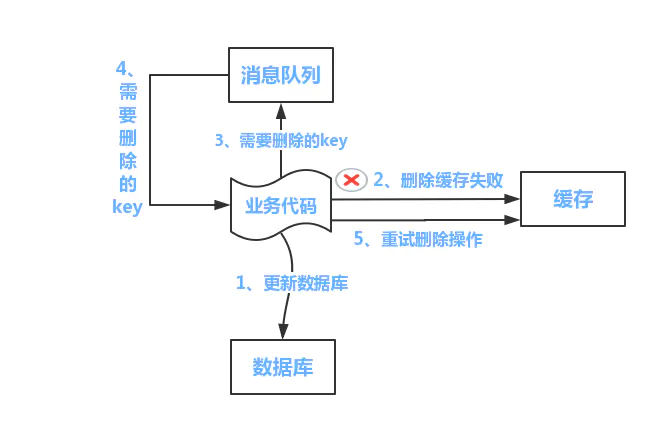
（6）数据库完成主从同步，从库变为新值

上述情形，就是数据不一致的原因。还是使用双删延时策略。只是，睡眠时间修改为在主从同步的延时时间基础上，加几百ms。

### 删缓存失败了怎么办：重试机制

看似问题都已经解决了，但其实，还有一个问题没有考虑到，那就是删除缓存的操作，失败了怎么办？比如延时双删的时候，第二次缓存删除失败了，那不还是没有清除脏数据吗？

解决方案就是再加上一个重试机制，保证删除缓存成功。



流程如下所示

（1）更新数据库数据；

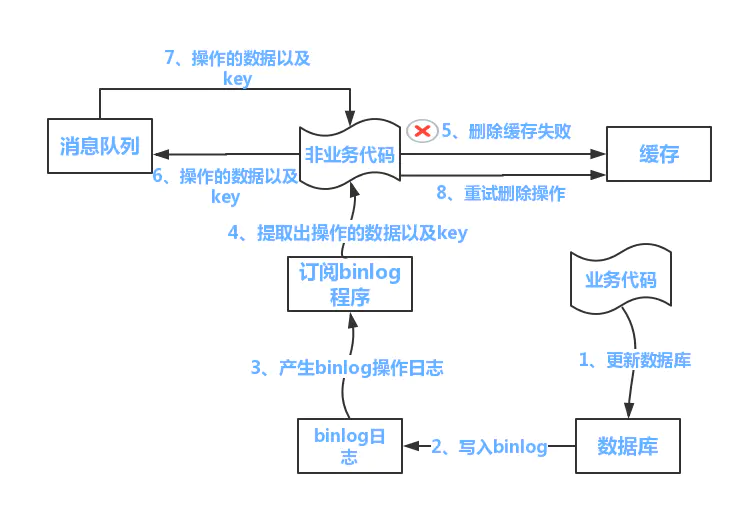
（2）缓存因为种种问题删除失败

（3）将需要删除的key发送至消息队列

（4）自己消费消息，获得需要删除的key

（5）继续重试删除操作，直到成功

然而，该方案有一个缺点，对业务线代码造成大量的侵入。于是有了方案二，在方案二中，启动一个订阅程序去订阅数据库的binlog，获得需要操作的数据。在应用程序中，另起一段程序，获得这个订阅程序传来的信息，进行删除缓存操作。



流程如下图所示：

（1）更新数据库数据

（2）数据库会将操作信息写入binlog日志当中

（3）订阅程序提取出所需要的数据以及key

（4）另起一段非业务代码，获得该信息

（5）尝试删除缓存操作，发现删除失败

（6）将这些信息发送至消息队列

（7）重新从消息队列中获得该数据，重试操作。

而读取binlog的中间件，可以采用阿里开源的cana

好了，到这里我们已经把缓存双写一致性的思路彻底梳理了一遍，下面就是我对这几种思路徒手写的实战代码，方便有需要的朋友参考。

# redis 的主从

## redis主从

### 引入

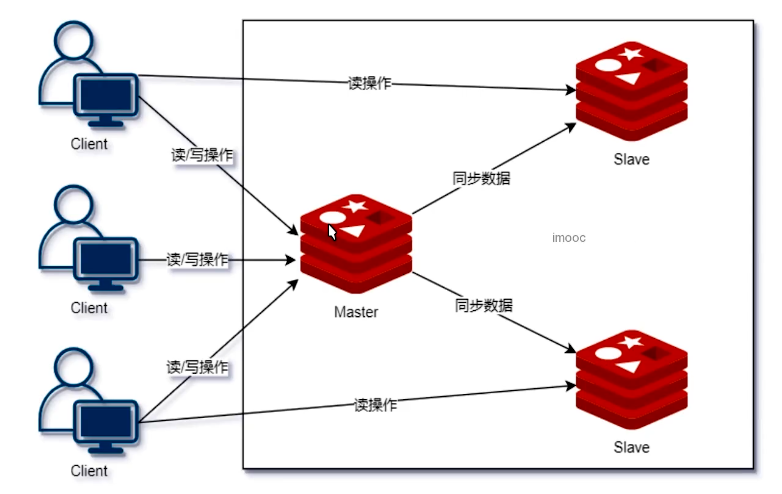
单机redis面临的问题:

1、机器故障。我们部署到一台 Redis 服务器，当发生机器故障时，需要迁移到另外一台服务器并且要保证数据是同步的。而数据是最重要的，如果你不在乎，基本上也就不会使用 Redis 了。

2、容量瓶颈。当我们有需求需要扩容 Redis 内存时，从 16G 的内存升到 64G，单机肯定是满足不了。当然，你可以重新买个 128G 的新机器。

3、性能瓶颈，读写都会打到单机上，redis的作用缓存，读的请求要比写请求大的多。

### 主从复制



主从复制，是指将一台Redis服务器的数据，复制到其他的Redis服务器。前者称为主节点(master)，后者称为从节点(slave)；数据的复制是单向的，只能由主节点到从节点。

默认情况下，每台Redis服务器都是主节点；且一个主节点可以有多个从节点(或没有从节点)，但一个从节点只能有一个主节点。

通过多个 slave，Redis 的节点数据就可以实现多副本保存，任何一个节点异常都不会导致数据丢失，同时多 slave 可以 N 倍提升读性能。master 只写不读，这样整个 master-slave 组合，读写能力都可以得到大幅提升。

master 在分发写请求时，同时会将写指令复制一份存入复制积压缓冲，这样当 slave 短时间断开重连时，只要 slave 的复制位置点仍然在复制积压缓冲，则可以从之前的复制位置点之后继续进行复制，提升复制效率。

### 主从作用

数据冗余：主从复制实现了数据的热备份，是持久化之外的一种数据冗余方式。

故障恢复：当主节点出现问题时，可以由从节点提供服务，实现快速的故障恢复；实际上是一种服务的冗余。

负载均衡：在主从复制的基础上，配合读写分离，可以由主节点提供写服务，由从节点提供读服务（即写Redis数据时应用连接主节点，读Redis数据时应用连接从节点），分担服务器负载；尤其是在写少读多的场景下，通过多个从节点分担读负载，可以大大提高Redis服务器的并发量。

高可用基石：除了上述作用以外，主从复制还是哨兵和集群能够实施的基础，因此说主从复制是Redis高可用的基础。

### 优缺点

优点:主从分离

1、写压力放到master,读压力放到slave，增加并发量。

2、方便去拓展，并发压力上来之后，可以增加从节点。

3、主备的自动切换。

缺点:

1、主节点的写压力。

2、redis 单线程受限机器性能。

### 主从持久化

如果采用了主从架构,那么建议必须开启master node的持久化!

不建议用slave node作为master node的数据热备,因为那样的话,如果你关掉naster的持久化,可能在master宕机重启的时候数据是空的,然后一经复制的，salve node数据也丢了。

## 主从复制流程

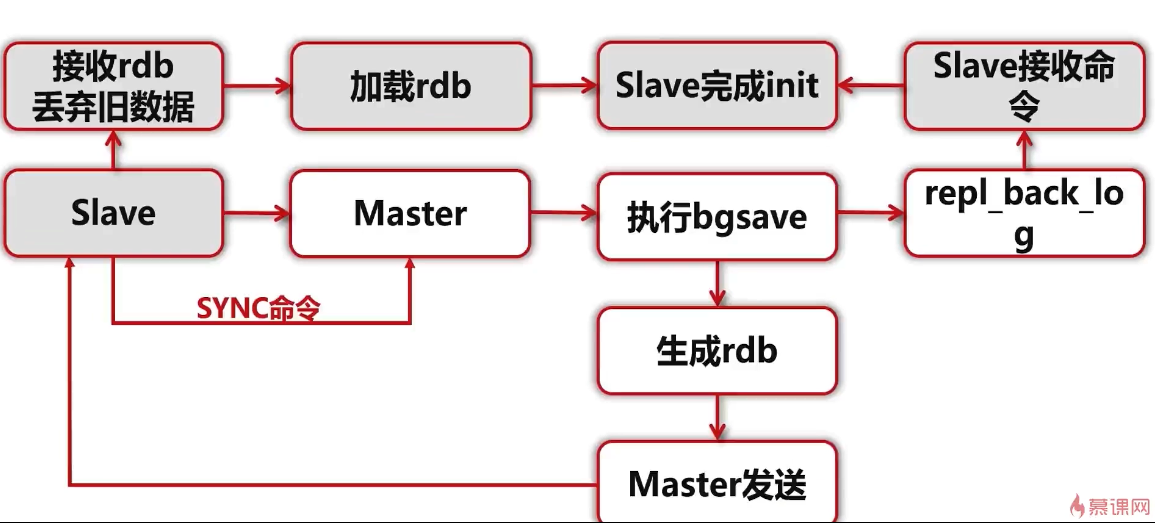
### 主从复制流程

全量复制场景:

1、集群启动

2、新的从节点的添加。

3、主节点故障，更换了主节点。



增量复制：

1、增量复制是slave初始化后开始正常工作时主服务器发生的写操作同步到从服务器的过程。

2、复制过程是主服务器每执行一个写命令就会向从服务器发送相同的写命令,从服务器接收并执行收到的写命令。

### 建立复制配置

需要注意，主从复制的开启，完全是在从节点发起的；不需要我们在主节点做任何事情，从节点开启主从复制，有3种方式：

（1）配置文件

在从服务器的配置文件中加入：slaveof <masterip> <masterport>

（2）启动命令

redis-server启动命令后加入 --slaveof <masterip> <masterport>

(3) 客户端命令

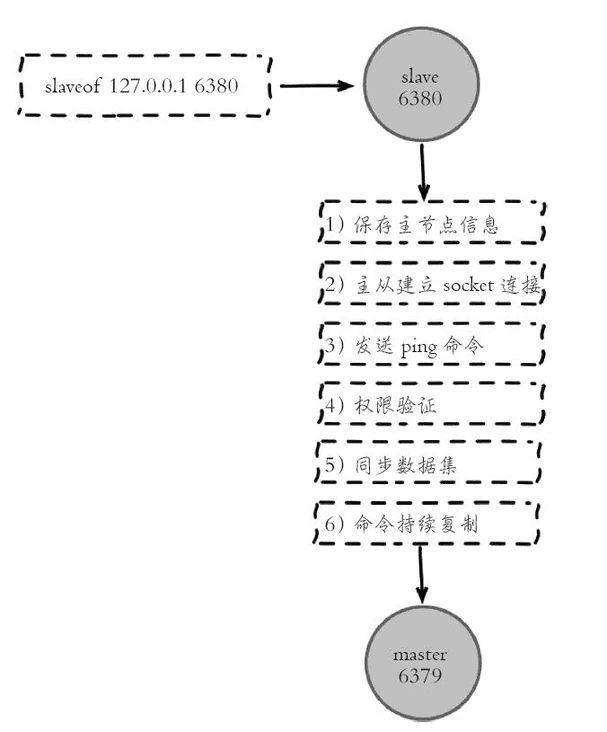
Redis服务器启动后，直接通过客户端执行命令：slaveof <masterip> <masterport>，则该Redis实例成为从节点。

上述3种方式是等效的，下面以客户端命令的方式为例，看一下当执行了slaveof后，Redis主节点和从节点的变化。

### 主从复制的原理

主从复制过程大体可以分为3个阶段：连接建立阶段（即准备阶段）、数据同步阶段、命令传播阶段。

在从节点执行 slaveof 命令后，复制过程便开始运作，下面图示大概可以看到，从图中可以看出复制过程大致分为6个过程

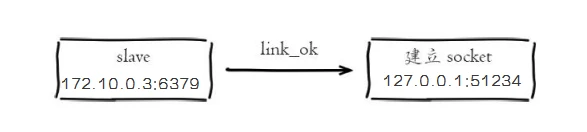


1、保存主节点（master）信息。

执行 slaveof 后 Redis 会打印如下日志：



2、从节点（slave）内部通过每秒运行的定时任务维护复制相关逻辑，当定时任务发现存在新的主节点后，会尝试与该节点建立网络连接。



从节点与主节点建立网络连接

从节点会建立一个 socket 套接字，从节点建立了一个端口为51234的套接字，专门用于接受主节点发送的复制命令。从节点连接成功后打印如下日志：



如果从节点无法建立连接，定时任务会无限重试直到连接成功或者执行 slaveof no one 取消复制

关于连接失败，可以在从节点执行 info replication 查看 master\_link\_down\_since\_seconds 指标，它会记录与主节点连接失败的系统时间。从节点连接主节点失败时也会每秒打印如下日志，方便发现问题：

*# Error condition on socket for SYNC: {socket\_error\_reason}*

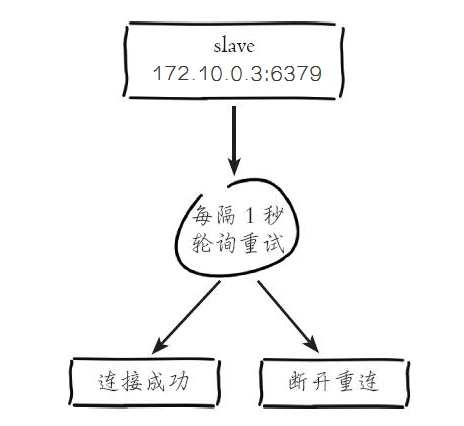
3）发送 ping 命令。

连接建立成功后从节点发送 ping 请求进行首次通信，ping 请求主要目的如下：

·检测主从之间网络套接字是否可用。

·检测主节点当前是否可接受处理命令。

如果发送 ping 命令后，从节点没有收到主节点的 pong 回复或者超时，比如网络超时或者主节点正在阻塞无法响应命令，从节点会断开复制连接，下次定时任务会发起重连。





从节点发送的 ping 命令成功返回，Redis 打印如下日志，并继续后续复制流程：



4）权限验证。如果主节点设置了 requirepass 参数，则需要密码验证，从节点必须配置 masterauth 参数保证与主节点相同的密码才能通过验证；如果验证失败复制将终止，从节点重新发起复制流程。

5）同步数据集。主从复制连接正常通信后，对于首次建立复制的场景，主节点会把持有的数据全部发送给从节点，这部分操作是耗时最长的步骤。=

6）命令持续复制。当主节点把当前的数据同步给从节点后，便完成了复制的建立流程。接下来主节点会持续地把写命令发送给从节点，保证主从数据一致性。

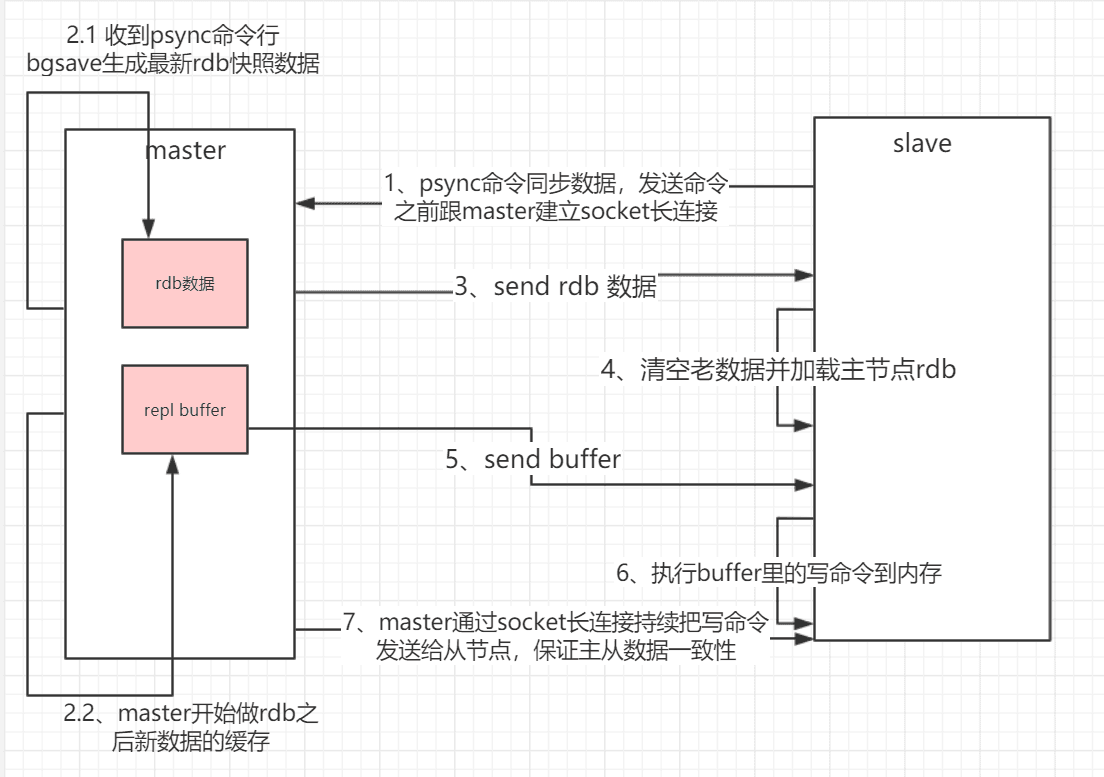
## 数据同步阶段

在Redis2.8以前，从节点向主节点发送sync命令请求同步数据，此时的同步方式是全量复制；在Redis2.8及以后，从节点可以发送psync命令请求同步数据，此时根据主从节点当前状态的不同，同步方式可能是全量复制或部分复制。后文介绍以Redis2.8及以后版本为例。

1、全量复制：用于初次复制或其他无法进行部分复制的情况，将主节点中的所有数据都发送给从节点，是一个非常重型的操作。、

2、部分复制：用于网络中断等情况后的复制，只将中断期间主节点执行的写命令发送给从节点，与全量复制相比更加高效。需要注意的是，如果网络中断时间过长，导致主节点没有能够完整地保存中断期间执行的写命令，则无法进行部分复制，仍使用全量复制。

### 全量复制



从节点判断无法进行部分复制，向主节点发送全量复制的请求；或从节点发送部分复制的请求，但主节点判断无法进行部分复制；

1、master 配置了一个 slave，slave 连接上 Master，就会发送一个 PSYNC 命令给 master 请求复制数据。

2、master 收到 PSYNC 命令后，在后台进行数据持久化，通过 bgsave 生成最新的 rdb 快照文件，持久化期间，master 会继续接收客户端的请求，会把这些可能修改数据集的请求缓存在内存中(写命令)。

3、当持久化进行完毕以后，master 会把这份 rdb 文件数据集发送给 slave

4、slave 会把接收到的数据进行持久化生成 rdb，然后再加载到内存中

5、master 再将之前缓存在内存中的命令发送给 slave。

6、当 master 与 slave 之间的连接由于某些原因而断开时，slave 能够自动重连 Master

7、如果 master 收到了多个 slave 并发连接请求，它只会进行一次持久化，而不是一个连接一次，再把这一份持久化的数据发送给多个并发连接的 slave

8、如果从节点开启了AOF，则会触发bgrewriteaof的执行，从而保证AOF文件更新至主节点的最新状态

通过全量复制的过程可以看出，全量复制是非常重型的操作：

1）主节点通过bgsave命令fork子进程进行RDB持久化，该过程是非常消耗CPU、内存(页表复制)、硬盘IO的；关于bgsave的性能问题，可以参考我另外一篇文章： 深入剖析Redis高可用系列：持久化 AOF和RDB

（2）主节点通过网络将RDB文件发送给从节点，对主从节点的带宽都会带来很大的消耗

（3）从节点清空老数据、载入新RDB文件的过程是阻塞的，无法响应客户端的命令；如果从节点执行bgrewriteaof，也会带来额外的消耗

### 部分复制

由于全量复制在主节点数据量较大时效率太低，因此Redis2.8开始提供部分复制，用于处理网络中断时的数据同步。

部分复制的实现，依赖于三个重要的概念：

1、复制偏移量

2、复制积压缓冲区

3、服务器运行ID(runid)

下面我们分别讲解一下这三个概念：

复制偏移量：

执行复制的双方，主从节点，分别会维护一个复制偏移量offset： 主节点每次向从节点同步了N字节数据后，将修改自己的复制偏移量offset+N 从节点每次从主节点同步了N字节数据后，将修改自己的复制偏移量offset+N

offset用于判断主从节点的数据库状态是否一致： 如果二者offset相同，则一致； 如果offset不同，则不一致，此时可以根据两个offset找出从节点缺少的那部分数据。

例如，如果主节点的offset是1000，而从节点的offset是500，那么部分复制就需要将offset为501-1000的数据传递给从节点。而offset为501-1000的数据存储的位置，就是下面要介绍的复制积压缓冲区。

复制积压缓冲区：

主节点内部维护了一个固定长度的、先进先出(FIFO)队列 作为复制积压缓冲区，其默认大小为1MB 在主节点进行命令传播时，不仅会将写命令同步到从节点，还会将写命令写入复制积压缓冲区。

由于复制积压缓冲区定长且是先进先出，所以它保存的是主节点最近执行的写命令；时间较早的写命令会被挤出缓冲区。因此，当主从节点offset的差距过大超过缓冲区长度时，将无法执行部分复制，只能执行全量复制。

为了提高网络中断时部分复制执行的概率，可以根据需要增大复制积压缓冲区的大小(通过配置repl-backlog-size)；例如如果网络中断的平均时间是60s，而主节点平均每秒产生的写命令(特定协议格式)所占的字节数为100KB，则复制积压缓冲区的平均需求为6MB，保险起见，可以设置为12MB，来保证绝大多数断线情况都可以使用部分复制。

从节点将offset发送给主节点后，主节点根据offset和缓冲区大小决定能否执行部分复制：

如果offset偏移量之后的数据，仍然都在复制积压缓冲区里，则执行部分复制；

如果offset偏移量之后的数据已不在复制积压缓冲区中（数据已被挤出），则执行全量复制。

复制积压缓冲区示意图：



服务器运行ID(runid)：

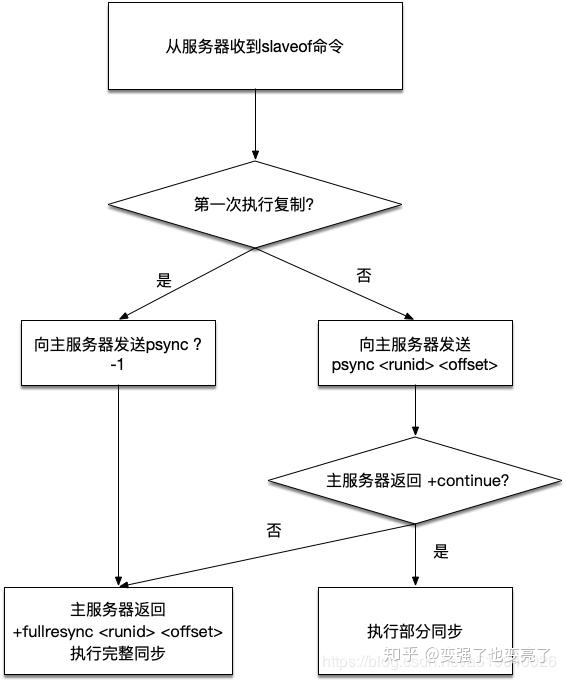
每个Redis节点，都有其运行ID，运行ID由节点在启动时自动生成，主节点会将自己的运行ID发送给从节点，从节点会将主节点的运行ID存起来。 从节点Redis断开重连的时候，就是根据运行ID来判断同步的进度：

如果从节点保存的runid与主节点现在的runid相同，说明主从节点之前同步过，主节点会继续尝试使用部分复制(到底能不能部分复制还要看offset和复制积压缓冲区的情况)；

如果从节点保存的runid与主节点现在的runid不同，说明从节点在断线前同步的Redis节点并不是当前的主节点，只能进行全量复制。

### psync 命令的执行

在了解了复制偏移量、复制积压缓冲区、节点运行id之后，本节将介绍psync命令的参数和返回值，从而说明psync命令执行过程中，主从节点是如何确定使用全量复制还是部分复制的。 psync命令流程图如下：



psync命令的大体流程如下：

如果从节点之前没有复制过任何主节点，或者之前执行过slaveof no one命令，从节点就会向主节点发送psync命令，请求主节点进行数据的全量同步

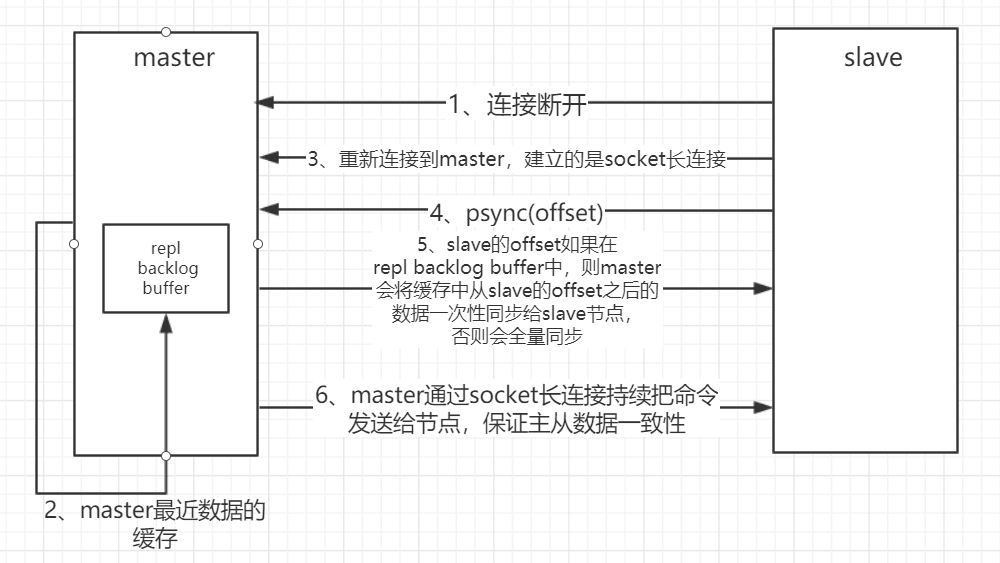
如果前面从节点已经同步过部分数据，此时从节点就会发送psync {runid} {offset}命令给主节点，其中runid是上一次主节点的运行ID，offset是当前从节点的复制偏移量

主节点收到psync命令后，会出现以下三种可能：

主节点返回 fullresync {runid} {offset}回复，表示主节点要求与从节点进行数据的完整全量复制，其中runid表示主节点的运行ID，offset表示当前主节点的复制偏移量

如果主服务器返回 +continue，表示主节点与从节点会进行部分数据的同步操作，将从服务器缺失的数据复制过来即可

如果主服务器返回 -err，表示主服务器的Redis版本低于2.8，无法识别psync命令，此时从服务器会向主服务器发送sync命令，进行完整的数据全量复制



1、当 master 和 slave 断开重连后，一般都会对整份数据进行复制。但从 redis2.8 版本开始，Redis 改用可以支持部分数据复制的命令 PSYNC 去 master 同步数据，slave 与 master 能够在网络连接断开重连后只进行部分数据复制(断点续传)。

2、master 在其内存中创建一个复制数据用的缓存队列，缓存最近一段时间的数据

3、master 和它所有的 slave 都维护了复制的数据下标 offset 和 master 的进程 id

4、因此，当网络连接断开后，slave 会请求 master 继续进行未完成的复制，从所记录的数据下标开始

5、如果 master 进程 id 变化了，或者从节点数据下标 offset 太旧，已经不在 master 的缓存队列里了，那么将会进行一次全量数据的复制。

## redis主从复制搭建

### 环境搭建

1、准备环境

|  |  |
| --- | --- |
| Ip | 角色 |
| 192.168.205.129 | Master |
| 192.168.205.128 | slave |
| 192.168.205.130 | slave |

2创建文件

mkdir –p /usr/local/redis/conf 配置文件

mkdir –p /usr/local/redis/data 数据目录

mkdir –p /usr/local/redis/log 日志文件

3、创建配置文件

cd /usr/local/redis/conf

master

|  |
| --- |
| #指定Redis只接收来自于该IP地址的请求,多个空格  bind 0.0.0.0  #默认情况下, redis不是在后台运行的,如果需要在后台运行,把该项的值更改为yes  daemonize yes  #日志文件  logfile "/usr/local/redis/log/redis.log"  #镜像备份文件的文件名。  dbfilename dump.rdb  #aop持久化开启  appendonly yes  #aop持久化文件名称  appendfilename "appendonly.aof"  #数据库镜像备份的文件放置的路径。  dir /usr/local/redis/data  #连接密码,没有密码,直接注释  requirepass 123456  #从节点访问主节点密码(必须与requirepass)  masterauth 123456  #从节点只读  replica-read-only yes |

Salve

|  |
| --- |
| #指定Redis只接收来自于该IP地址的请求,多个空格  bind 0.0.0.0  #默认情况下, redis不是在后台运行的,如果需要在后台运行,把该项的值更改为yes  daemonize yes  #日志文件  logfile "/usr/local/redis/log/redis.log"  #镜像备份文件的文件名。  dbfilename dump.rdb  #aop持久化开启  appendonly yes  #aop持久化文件名称  appendfilename "appendonly.aof"  #数据库镜像备份的文件放置的路径。  dir /usr/local/redis/data  #连接密码,没有密码,直接注释  requirepass 123456  #从节点访问主节点密码(必须与requirepass)  masterauth 123456  #从节点只读  replica-read-only yes  #从节点属于那个主节点  slaveof 192.168.205.129 6379 |

测试运行

/usr/local/redis/bin/redis-server /usr/local/redis/conf/redis.conf

ps -ef|grep redis

### 查看主从信息

/usr/local/redis/bin/redis-cli -a 123456 客户端连接

info replication 查看主从信息。

**Master**

|  |
| --- |
| role:master 角色  connected\_slaves:2 连接点从节点  slave0:ip=192.168.205.128,port=6379,state=online,offset=238,lag=0  slave1:ip=192.168.205.130,port=6379,state=online,offset=238,lag=0  master\_replid:a4a80951270b0ea2fe93b4e4cf128ae63d8cb45d  master\_replid2:0000000000000000000000000000000000000000  master\_repl\_offset:238  second\_repl\_offset:-1  repl\_backlog\_active:1 缓冲区开启  repl\_backlog\_size:1048576 缓冲区大小  repl\_backlog\_first\_byte\_offset:1  repl\_backlog\_histlen:238 |

Lag:延迟量，单位时间秒

主机写入redis命令，从机读取执行。主从的数据是否一致。

Offset:偏移量,从机已经复制的偏移量。

master\_repl\_offset:主节点写入的偏移量。

master\_replid:主节点id

master\_replid2:哨兵模式下如果主节点挂了之后，新的主节点给了master\_replid，旧的主节点赋给改值。

全量复制:

1、集群启动

2、新的从节点的添加。

3、主节点故障，更换了主节点。

second\_repl\_offset:主节点故障之后会将master\_repl\_offset+1赋给该值。作为新的主节点与旧主节点做一个增量同步。

**Slave:没有写权限。**

|  |
| --- |
| role:slave 角色  master\_host:192.168.205.129  master\_port:6379  master\_link\_status:up  master\_last\_io\_seconds\_ago:6  master\_sync\_in\_progress:0  slave\_repl\_offset:448  slave\_priority:100  slave\_read\_only:1  connected\_slaves:0  master\_replid:a4a80951270b0ea2fe93b4e4cf128ae63d8cb45d  master\_replid2:0000000000000000000000000000000000000000  master\_repl\_offset:448  second\_repl\_offset:-1  repl\_backlog\_active:1  repl\_backlog\_size:1048576  repl\_backlog\_first\_byte\_offset:1  repl\_backlog\_histlen:448 |

master\_last\_io\_seconds\_ago:主从上一次同步时间

master\_sync\_in\_progress:0：主从同步状态，1为正在同步

master\_repl\_offset:主节点写入偏移量

slave\_repl\_offset:448 :从节点复制的偏移量

slave\_priority:100 该从节点升为主节点的几率。

slave\_read\_only:1

connected\_slaves:0 该结点连接的从节点信息。

### master复制日志

Ready to accept connections

//从机的全量复制请求。

Replica 192.168.205.128:6379 asks for synchronization

Full resync requested by replica 192.168.205.128:6379

Replication backlog created, my new replication IDs are 'a4a80951270b0ea2fe93b4e4cf128ae63d8cb45d' and '0000000000000000000000000000000000000000'

//RDB操作

Starting BGSAVE for SYNC with target: disk

Background saving started by pid 1467

DB saved on disk

RDB: 0 MB of memory used by copy-on-write

Background saving terminated with success

//从机的全量复制结束。

Synchronization with replica 192.168.205.128:6379 succeeded

//从机的全量复制请求。

Replica 192.168.205.130:6379 asks for synchronization

Full resync requested by replica 192.168.205.130:6379

Starting BGSAVE for SYNC with target: disk

Background saving started by pid 1468

DB saved on disk

RDB: 0 MB of memory used by copy-on-write

Background saving terminated with success

Synchronization with replica 192.168.205.130:6379 succeeded

## 主从复制的问题

在主从复制基础上实现的读写分离，可以实现Redis的读负载均衡：由主节点提供写服务，由一个或多个从节点提供读服务（多个从节点既可以提高数据冗余程度，也可以最大化读负载能力）；在读负载较大的应用场景下，可以大大提高Redis服务器的并发量。下面介绍在使用Redis读写分离时，需要注意的问题。

### 延迟与不一致问题

延迟与不一致问题 前面已经讲到，由于主从复制的命令传播是异步的，延迟与数据的不一致不可避免。如果应用对数据不一致的接受程度程度较低，可能的优化措施包括：优化主从节点之间的网络环境（如在同机房部署）；监控主从节点延迟（通过offset）判断，如果从节点延迟过大，通知应用不再通过该从节点读取数据；使用集群同时扩展写负载和读负载等。

在命令传播阶段以外的其他情况下，从节点的数据不一致可能更加严重，例如连接在数据同步阶段，或从节点失去与主节点的连接时等。从节点的slave-serve-stale-data参数便与此有关：它控制这种情况下从节点的表现；如果为yes（默认值），则从节点仍能够响应客户端的命令，如果为no，则从节点只能响应info、slaveof等少数命令。该参数的设置与应用对数据一致性的要求有关；如果对数据一致性要求很高，则应设置为no。

### 数据过期问题

数据过期问题 在单机版Redis中，存在两种删除策略：

惰性删除：服务器不会主动删除数据，只有当客户端查询某个数据时，服务器判断该数据是否过期，如果过期则删除。 定期删除：服务器执行定时任务删除过期数据，但是考虑到内存和CPU的折中（删除会释放内存，但是频繁的删除操作对CPU不友好），该删除的频率和执行时间都受到了限制。 在主从复制场景下，为了主从节点的数据一致性，从节点不会主动删除数据，而是由主节点控制从节点中过期数据的删除。由于主节点的惰性删除和定期删除策略，都不能保证主节点及时对过期数据执行删除操作，因此，当客户端通过Redis从节点读取数据时，很容易读取到已经过期的数据。

Redis 3.2中，从节点在读取数据时，增加了对数据是否过期的判断：如果该数据已过期，则不返回给客户端；将Redis升级到3.2可以解决数据过期问题。

### 故障切换问题

在没有使用哨兵的读写分离场景下，应用针对读和写分别连接不同的Redis节点；当主节点或从节点出现问题而发生更改时，需要及时修改应用程序读写Redis数据的连接；连接的切换可以手动进行，或者自己写监控程序进行切换，但前者响应慢、容易出错，后者实现复杂，成本都不算低。

### 复制超时问题

主从节点复制超时是导致复制中断的最重要的原因之一，本小节单独说明超时问题，下一小节说明其他会导致复制中断的问题。

超时判断意义

在复制连接建立过程中及之后，主从节点都有机制判断连接是否超时，其意义在于：

1、如果主节点判断连接超时，其会释放相应从节点的连接，从而释放各种资源，否则无效的从节点仍会占用主节点的各种资源（输出缓冲区、带宽、连接等）；此外连接超时的判断可以让主节点更准确的知道当前有效从节点的个数，有助于保证数据安全（配合前面讲到的min-slaves-to-write等参数）

2、如果从节点判断连接超时，则可以及时重新建立连接，避免与主节点数据长期的不一致。

判断机制

主从复制超时判断的核心，在于repl-timeout参数，该参数规定了超时时间的阈值（默认60s），对于主节点和从节点同时有效；主从节点触发超时的条件分别如下：

主节点：每秒1次调用复制定时函数replicationCron()，在其中判断当前时间距离上次收到各个从节点REPLCONF ACK的时间，是否超过了repl-timeout值，如果超过了则释放相应从节点的连接。

从节点：从节点对超时的判断同样是在复制定时函数中判断，基本逻辑是： 如果当前处于连接建立阶段，且距离上次收到主节点的信息的时间已超过repl-timeout，则释放与主节点的连接； 如果当前处于数据同步阶段，且收到主节点的RDB文件的时间超时，则停止数据同步，释放连接； 如果当前处于命令传播阶段，且距离上次收到主节点的PING命令或数据的时间已超过repl-timeout值，则释放与主节点的连接。

### 主从的key问题

Slave不会让KEY过期,而是等待Master让KEY过期。当Master让KEY到期时,它会合成一个DEL命令并传输到所有的Slave.

加速复制

默认情况下, master节点接收SYNC命令后执行BGSAVE操作,将数据先保存到磁盘,如果磁盘性能差,那么写入磁盘会消耗大量性能,因此在Redis 2.8.18时进行改进,可以设置无需写入磁盘直接发生RDB快照给slave,加快复制速度。

修改配置: repl-diskless-sync yes (默认no)

### 实际问题

数据同步阶段：在主从节点进行全量复制bgsave时，主节点需要首先fork子进程将当前数据保存到RDB文件中，然后再将RDB文件通过网络传输到从节点。如果RDB文件过大，主节点在fork子进程+保存RDB文件时耗时过多，可能会导致从节点长时间收不到数据而触发超时；此时从节点会重连主节点，然后再次全量复制，再次超时，再次重连……这是个悲伤的循环。为了避免这种情况的发生，除了注意Redis单机数据量不要过大，另一方面就是适当增大repl-timeout值，具体的大小可以根据bgsave耗时来调整。

命令传播阶段：如前所述，在该阶段主节点会向从节点发送PING命令，频率由repl-ping-slave-period控制；该参数应明显小于repl-timeout值(后者至少是前者的几倍)。否则，如果两个参数相等或接近，网络抖动导致个别PING命令丢失，此时恰巧主节点也没有向从节点发送数据，则从节点很容易判断超时。

慢查询导致的阻塞：如果主节点或从节点执行了一些慢查询（如keys \*或者对大数据的hgetall等），导致服务器阻塞；阻塞期间无法响应复制连接中对方节点的请求，可能导致复制超时。

# 哨兵模式

## Redis哨兵

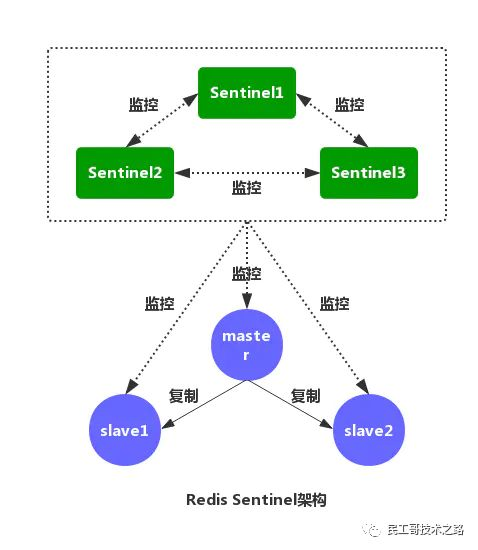
### 主从模式的缺点

1、主节点故障，无法写入数据，导致查询不可以用,查询都达到了mysql，导致宕机。

2、主节点的写压力。

3、主从不具备自动的主备切换，故障之后需要手动切将slave切换为master,设置为可写，并且还有通知客户端的切换。

### 哨兵模式



Sentinel(哨兵)是用于监控redis集群中Master状态的工具，是Redis 的高可用性解决方案，sentinel哨兵模式已经被集成在redis2.4之后的版本中。sentinel是redis高可用的解决方案，sentinel系统可以监视一个或者多个redis master服务，以及这些master服务的所有从服务；当某个master服务下线时，自动将该master下的某个从服务升级为master服务替代已下线的master服务继续处理请求。

sentinel可以让redis实现主从复制，当一个集群中的master失效之后，sentinel可以选举出一个新的master用于自动接替master的工作，集群中的其他redis服务器自动指向新的master同步数据。一般建议sentinel采取奇数台，防止某一台sentinel无法连接到master导致误切换。

当用Redis做Master-slave的高可用方案时，假如master宕机了，Redis本身(包括它的很多客户端)都没有实现自动进行主备切换，而Redis-sentinel本身也是一个独立运行的进程，它能监控多个master-slave集群，发现master宕机后能进行自动切换。Sentinel由一个或多个Sentinel 实例 组成的Sentinel 系统可以监视任意多个主服务器，以及这些主服务器属下的所有从服务器，并在被监视的主服务器进入下线状态时，自动将下线主服务器属下的某个从服务器升级为新的主服务器。

### 哨兵模式的作用

监控：哨兵会不断检查你的主服务器和从服务器是否运作正常。

提醒：当被监控的某个Redis服务器出现问题时，哨兵可以通过API给程序员发送通知。

自动故障转移：主服务器宕机，哨兵会开始一次自动故障转移操作，升级一个从服务器为主服务器，并让其他从服务器改为复制新的主服务器.

### 哨兵的配置

1）sentinel monitor <masterName> <ip> <port> <quorum>

四个参数含义：

masterName这个是对某个master+slave组合的一个区分标识（一套sentinel是可以监听多套master+slave这样的组合的）。

ip 和 port 就是master节点的 ip 和 端口号。

quorum这个参数是进行客观下线的一个依据，意思是至少有 quorum 个sentinel主观的认为这个master有故障，才会对这个master进行下线以及故障转移。因为有的时候，某个sentinel节点可能因为自身网络原因，导致无法连接master，而此时master并没有出现故障，所以这就需要多个sentinel都一致认为该master有问题，才可以进行下一步操作，这就保证了公平性和高可用。

2）sentinel down-after-milliseconds <masterName> <timeout>

这个配置其实就是进行主观下线的一个依据，masterName这个参数不用说了，timeout是一个毫秒值，表示：如果这台sentinel超过timeout这个时间都无法连通master包括slave（slave不需要客观下线，因为不需要故障转移）的话，就会主观认为该master已经下线（实际下线需要客观下线的判断通过才会下线）

那么，多个sentinel之间是如何达到共识的呢？

这就是依赖于前面说的第二个定时任务，某个sentinel先将master节点进行一个主观下线，然后会将这个判定通过sentinel is-master-down-by-addr这个命令问对应的节点是否也同样认为该addr的master节点要做客观下线。最后当达成这一共识的sentinel个数达到前面说的quorum设置的这个值时，就会对该master节点下线进行故障转移。quorum的值一般设置为sentinel个数的二分之一加1，例如3个sentinel就设置2。

。

### 哨兵的优缺点

1、哨兵模式是基于主从模式的,所有主从的优点,哨兵模式都有;

2、主从可以自动切换,系统更健壮,可用性更高;

3、Sentinel会不断地检查你的主服务器和从服务器是否运作正常。当被监控的某个Redis服务器出现问题时, Sentinel可以通过API向管理员或者其他应用程序发送通知。

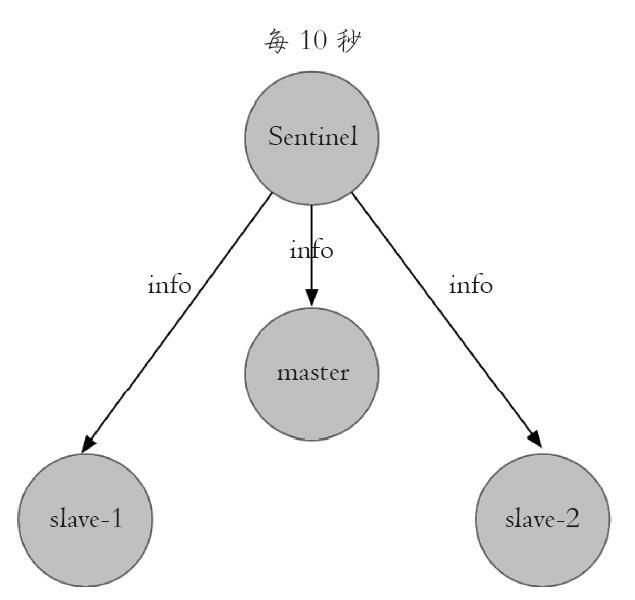
### sentinel 的定时任务

sentinel在内部有3个定时任务

1）每10秒每个sentinel会对master和slave执行info命令，这个任务达到两个目的：

a）发现slave节点

b）确认主从关系



2）每2秒每个sentinel通过master节点的channel交换信息（pub/sub）。master节点上有一个发布订阅的频道(sentinel:hello)。sentinel节点通过sentinel:hello频道进行信息交换(对节点的"看法"和自身的信息)，达成共识。

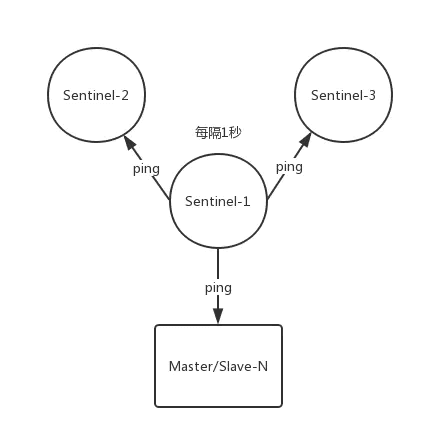


发现新的Sentinel节点： 通过订阅主节点的sentinel： hello了解其他的Sentinel节点信息， 如果是新加入的Sentinel节点， 将该Sentinel节点信息保存起来， 并与该Sentinel节点创建连接

·Sentinel节点之间交换主节点的状态， 作为后面客观下线以及领导者选举的依据。

|  |
| --- |
| 127.0.0.1:6379> pubsub channels  1) "\_\_sentinel\_\_:hello"  127.0.0.1:6379> subscribe \_\_sentinel\_\_:hello  Reading messages... (press Ctrl-C to quit)  1) "subscribe"  2) "\_\_sentinel\_\_:hello"  3) (integer) 1  1) "message"  2) "\_\_sentinel\_\_:hello"  3) "192.168.179.104,26379,4adf7ee7492961131ffcd0d40d8683fc11ae2294,0,mymaster,192.168.179.102,6379,0"  1) "message"  2) "\_\_sentinel\_\_:hello"  3) "192.168.179.102,26379,bf889f09dcbd3539e41d06b575fc831dce9c0538,0,mymaster,192.168.179.102,6379,0"  1) "message"  2) "\_\_sentinel\_\_:hello"  3) "192.168.179.103,26379,2fe22e2d94cc8eea00e13dffafd7563a742147a1,0,mymaster,192.168.179.102,6379,0" |

3）每1秒每个sentinel对其他sentinel和redis节点执行ping操作（相互监控），这个其实是一个心跳检测，是失败判定的依据。



### Sentinel发现

一个 Sentinel 可以与其他多个 Sentinel 进行连接， 各个 Sentinel 之间可以互相检查对方的可用性，并进行信息交换。

你无须为运行的每个 Sentinel 分别设置其他 Sentinel 的地址，因为 Sentinel 可以通过发布与订阅功能来自动发现正在监视相同主服务器的其他 Sentinel。

1、每个 Sentinel 会以每两秒一次的频率，通过发布与订阅功能，向被它监视的所有主服务器和从服务器的频道发送一条信息，信息中包含了 Sentinel 的 IP 地址、端口号和运行 ID （runid）。

2、每个 Sentinel 都订阅了被它监视的所有主服务器和从服务器的频道， 查找之前未出现过的 sentinel 。当一个 Sentinel 发现一个新的 Sentinel 时， 它会将新的 Sentinel 添加到一个列表中。

3、Sentinel 发送的信息中还包括完整的主服务器当前配置。如果一个 Sentinel 包含的主服务器配置比另一个 Sentinel 发送的配置要旧， 那么这个 Sentinel 会立即升级到新配置上。

4、在将一个新 Sentinel 添加到监视主服务器的列表上面之前， Sentinel 会先检查列表中是否已经包含了和要添加的 Sentinel 拥有相同运行 ID 或者相同地址（包括 IP 地址和端口号）的 Sentinel ， 如果是的话， Sentinel 会先移除列表中已有的那些拥有相同运行 ID 或者相同地址的 Sentinel ， 然后再添加新 Sentinel

### redis 的下线

**主观下线:**所谓主观下线（Subjectively Down， 简称 SDOWN）指的是单个Sentinel实例对服务器做出的下线判断，即单个sentinel认为某个服务下线（有可能是接收不到订阅，之间的网络不通等等原因）。

主观下线就是说如果服务器在down-after-milliseconds给定的毫秒数之内， 没有返回 Sentinel 发送的 PING 命令的回复， 或者返回一个错误， 那么 Sentinel 将这个服务器标记为主观下线（SDOWN ）。

sentinel会以每秒一次的频率向所有与其建立了命令连接的实例（master，从服务，其他sentinel）发ping命令，通过判断ping回复是有效回复，还是无效回复来判断实例时候在线（对该sentinel来说是“主观在线”）。

sentinel配置文件中的down-after-milliseconds设置了判断主观下线的时间长度，如果实例在down-after-milliseconds毫秒内，返回的都是无效回复，那么sentinel回认为该实例已（主观）下线，修改其flags状态为SRI\_S\_DOWN。如果多个sentinel监视一个服务，有可能存在多个sentinel的down-after-milliseconds配置不同，这个在实际生产中要注意。

**客观下线**:客观下线（Objectively Down， 简称 ODOWN）指的是多个 Sentinel 实例在对同一个服务器做出 SDOWN 判断， 并且通过 SENTINEL is-master-down-by-addr 命令互相交流之后， 得出的服务器下线判断，然后开启failover。

客观下线就是说只有在足够数量的 Sentinel 都将一个服务器标记为主观下线之后， 服务器才会被标记为客观下线（ODOWN）。

只有当master被认定为客观下线时，才会发生故障迁移。

当sentinel监视的某个服务主观下线后，sentinel会询问其它监视该服务的sentinel，看它们是否也认为该服务主观下线，接收到足够数量（这个值可以配置）的sentinel判断为主观下线，既任务该服务客观下线，并对其做故障转移操作。

sentinel通过发送 SENTINEL is-master-down-by-addr ip port current\_epoch runid，（ip：主观下线的服务id，port：主观下线的服务端口，current\_epoch：sentinel的纪元，runid：\*表示检测服务下线状态，如果是sentinel 运行id，表示用来选举领头sentinel）来询问其它sentinel是否同意服务下线。

一个sentinel接收另一个sentinel发来的is-master-down-by-addr后，提取参数，根据ip和端口，检测该服务时候在该sentinel主观下线，并且回复is-master-down-by-addr，回复包含三个参数：down\_state（1表示已下线，0表示未下线），leader\_runid（领头sentinal id），leader\_epoch（领头sentinel纪元）。

sentinel接收到回复后，根据配置设置的下线最小数量，达到这个值，既认为该服务客观下线。

客观下线条件只适用于主服务器： 对于任何其他类型的 Redis 实例， Sentinel 在将它们判断为下线前不需要进行协商， 所以从服务器或者其他 Sentinel 永远不会达到客观下线条件。只要一个 Sentinel 发现某个主服务器进入了客观下线状态， 这个 Sentinel 就可能会被其他 Sentinel 推选出， 并对失效的主服务器执行自动故障迁移操作。

### redis哨兵产生的数据丢失

entinal（哨兵）模式+Redis主从复制

1、异步复制导致的数据丢失

主节点宕机后，在Sentinal（哨兵）未检测到并执行主从切换之前快速重启（如服务器上有自动重启程序，一旦发现Redis宕机立即重启进程），如主节点没有做数据持久化，那么主节点重启后，因为主节点run id发生变化，主从节点的数据将会全部丢失，如主节点做了数据持久化，那么主节点重启后，只会丢失宕机前没有持久化的那部分数据，只是所有的从节点都需要做一次全量复制

主节点宕机后，Sentinal（哨兵）集群检测到了主节点宕机并且已经从从节点中选出了一个主节点，完成了主节点切换。宕机的节点重启后，Sentinal（哨兵）会自动将该节点强制转换为新选举出来的主节点的从节点，并从新的那个主节点同步数据，那么这种情况下，会丢失宕机前未复制到新的主节点（原来的一个从节点）上的那部分数据

2、集群脑裂

什么是集群脑裂？：集群脑裂指的就是因为网络原因，主节点A与从节点、哨兵节点之间断开了链接，但是还在运行，但是哨兵节点与主节点A间已经失去了心跳，认为主节点A宕机，哨兵节点在所有从节点中进行故障恢复，选出了一个新的主节点B继续对外提供服务，导致整个集群中有两个主节点。连接到旧的主节点A的客户端还在继续向该节点写数据，连接到新的主节点B的客户端也持续向新的主节点B写数据。造成同一个集群，有两个写入节点的情况。

那么在集群脑裂的情况下，因为读写分离的设计，主节点负责写数据，从节点负责读数据，写入主节点A的数据，在从节点根本无法读出来。节点A重启之后，会成为节点B的从节点，那么从节点A与从节点、哨兵节点之间断开连接之后，写入节点A的数据就会丢失。

**减少数据丢失解决方案:**

**1、redis控制**

**min-slaves-to-write n**

**min-slaves-max-lag s**

**这组命令表达的是，至少要保证n个从节点从主节点复制数据的延迟不能超过s秒钟，如n设置为1，s设置为10，就表示集群中至少要有1个从几点从主节点复制数据的延迟不能超过10秒钟，一旦所有的节点，从主节点同步数据的时间延迟都超过10秒，主节点停止接收任何写请求。**

这样就能保证，即使因为脑裂，还是一部复制数据延迟情况下，数据丢失最多就丢失s秒的数据。

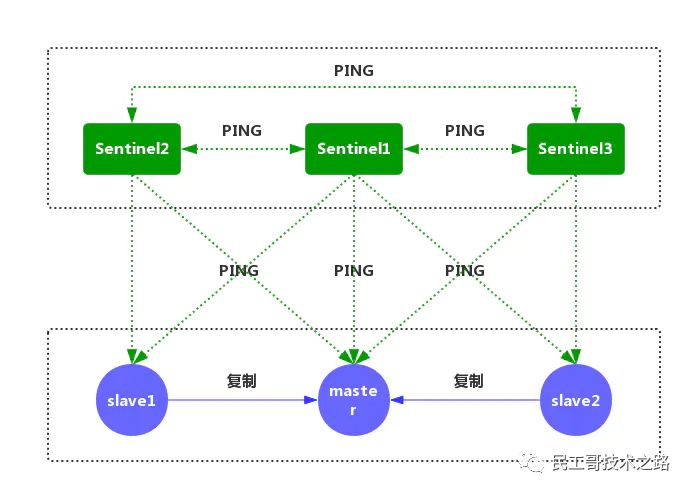
2、client端控制

一般来说，会在client做降级,写到本地磁盘里面，在client对外接收请求,再做降级，做限流,减慢请求涌入的速度。

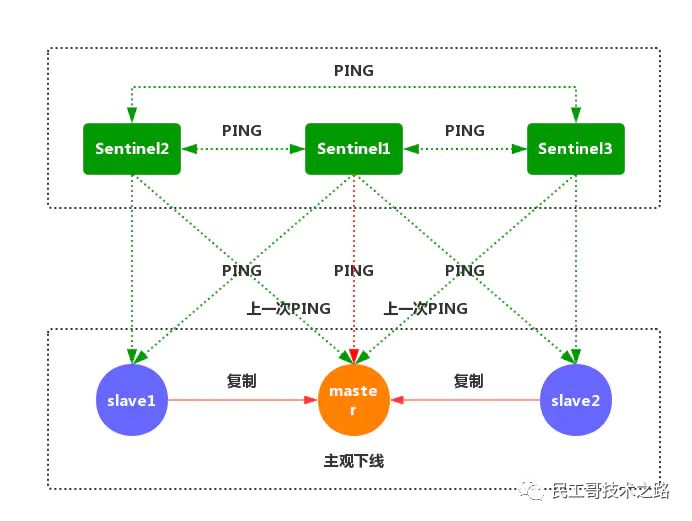
或者client可能会采取将数据临时灌入-个kafka消息队列,每隔10分钟去队列里面取一次,尝试重新发回master。

## Redis Sentinel的工作原理

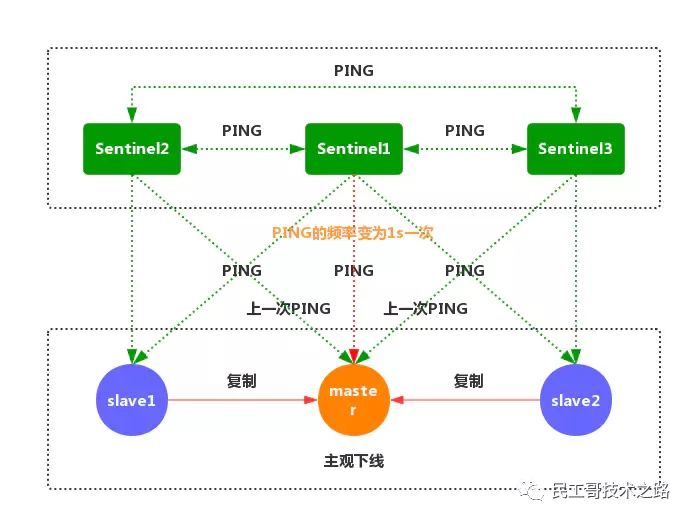
1.每个 Sentinel 以每秒一次的频率向它所知的主服务器、从服务器以及其他 Sentinel 实例发送一个 PING 命令。



2.如果一个实例距离最后一次有效回复 PING 命令的时间超过指定的值， 那么这个实例会被 Sentinel 标记为主观下线。

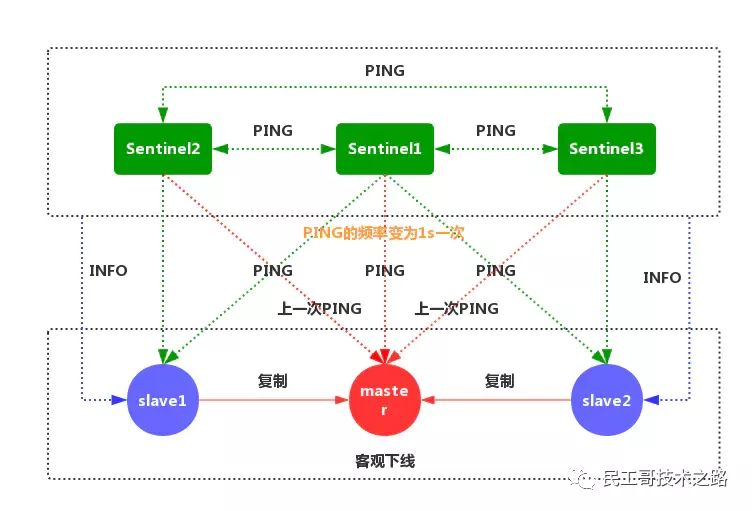


3.正在监视这个主服务器的所有 Sentinel 要以每秒一次的频率确认主服务器的确进入了主观下线状态。

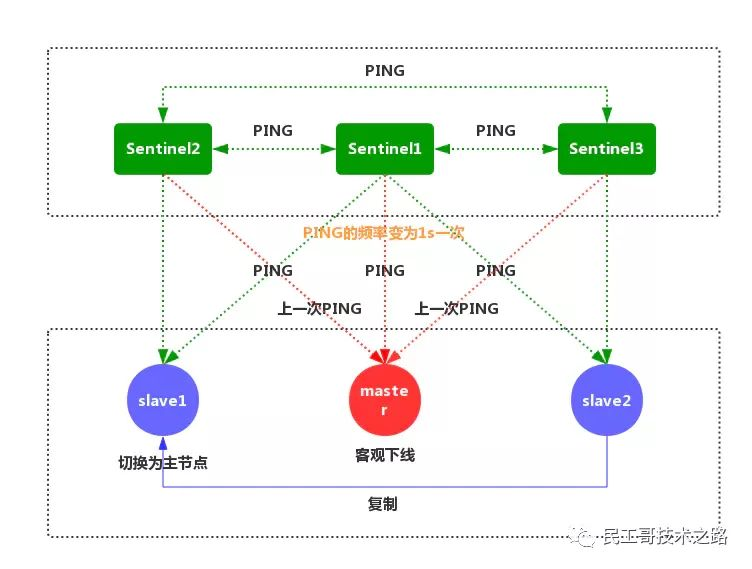


4.有足够数量的 Sentinel 在指定的时间范围内同意这一判断， 那么这个主服务器被标记为客观下线。

5.每个 Sentinel 会以每 10 秒一次的频率向它已知的所有主服务器和从服务器发送 INFO 命令。当一个主服务器被 Sentinel 标记为客观下线时， Sentinel 向下线主服务器的所有从服务器发送 INFO 命令的频率会从 10 秒一次改为每秒一次。



6.Sentinel 和其他 Sentinel 协商主节点的状态，如果主节点处于 SDOWN 状态，则投票自动选出新的主节点。将剩余的从节点指向新的主节点进行数据复制。



7.当没有足够数量的 Sentinel 同意主服务器 下线时， 主服务器的客观下线状态就会被移除。当主服务器重新向 Sentinel 的 PING 命令返回有效回复时，主服务器的主观下线状态就会被移除。

## sentinel的选举

前面我们谈到，当一个master被sentinel集群监控时，需要为它指定一个参数，这个参数指定了当需要判决master为不可用，并且进行failover时，所需要的sentinel数量，可以称这个参数为票数

不过，当failover主备切换真正被触发后，failover并不会马上进行，还需要sentinel中的大多数sentinel授权后才可以进行failover。

当ODOWN时，failover被触发。failover一旦被触发，尝试去进行failover的sentinel会去获得“大多数”sentinel的授权（如果票数比大多数还要大的时候，则询问更多的sentinel)

这个区别看起来很微妙，但是很容易理解和使用。例如，集群中有5个sentinel，票数被设置为2，当2个sentinel认为一个master已经不可用了以后，将会触发failover，但是，进行failover的那个sentinel必须先获得至少3个sentinel的授权才可以实行failover。

如果票数被设置为5，要达到ODOWN状态，必须所有5个sentinel都主观认为master为不可用，要进行failover，那么得获得所有5个sentinel的授权。

一个redis服务被判断为客观下线时，多个监视该服务的sentinel协商，选举一个领头sentinel，对该redis服务进行故障转移操作。选举领头sentinel遵循以下规则：

1）所有的sentinel都有公平被选举成领头的资格。

2）所有的sentinel都有且只有一次将某个sentinel选举成领头的机会（在一轮选举中），一旦选举某个sentinel为领头，不能更改。

3）sentinel设置领头sentinel是先到先得，一旦当前sentinel设置了领头sentinel，以后要求设置sentinel为领头请求都会被拒绝。

4）每个发现服务客观下线的sentinel，都会要求其他sentinel将自己设置成领头。

5）当一个sentinel（源sentinel）向另一个sentinel（目sentinel）发送is-master-down-by-addr ip port current\_epoch runid命令的时候，runid参数不是\*，而是sentinel运行id，就表示源sentinel要求目标sentinel选举其为领头。

6）源sentinel会检查目标sentinel对其要求设置成领头的回复，如果回复的leader\_runid和leader\_epoch为源sentinel，表示目标sentinel同意将源sentinel设置成领头。

7）如果某个sentinel被半数以上的sentinel设置成领头，那么该sentinel既为领头。

8）如果在限定时间内，没有选举出领头sentinel，暂定一段时间，再选举

选举过程：

1）每个做主观下线的sentinel节点向其他sentinel节点发送上面那条命令，要求将它设置为领导者。

2）收到命令的sentinel节点如果还没有同意过其他的sentinel发送的命令（还未投过票），那么就会同意，否则拒绝。

3）如果该sentinel节点发现自己的票数已经过半且达到了quorum的值，就会成为领导者

4）如果这个过程出现多个sentinel成为领导者，则会等待一段时间重新选举。

## 搭建哨兵模式

1、准备环境

|  |  |
| --- | --- |
| Ip | 角色 |
| 192.168.205.129 | Master |
| 192.168.205.128 | slave |
| 192.168.205.130 | slave |

2创建文件配置文件

创建配置文件sentinel.conf 放到 conf文件夹下。

|  |
| --- |
| #指定Redis只接收来自于该IP地址的请求,多个空格  bind 0.0.0.0  #进程端口号  port 6379  #默认情况下, redis不是在后台运行的,如果需要在后台运行,把该项的值更改为yes  daemonize yes  #日志文件  logfile "/usr/local/redis/log/sentinel.log"  #进程编号记录文件  pidfile /var/run/sentinel.pid  #指示sentinel去件事一个名为mymaster的主服务器，三台机器,两台同意  sentinel monitor mymaster 192.168.205.129 6379 2  #访问主节点的密码  sentinel auth-pass mymaster 123456  # sentinel 认为服务器已经断线所需要的毫秒数。ping之后返回的pong的间隔时间。  sentinel down-after-milliseconds mymaster 10000  #若sentinel在改配置内未完成failover(故障迁移)操作，则认为本次的failover失败。  sentinel failover-timeout mymaster 180000 |

3、启动服务

/usr/local/redis/bin/redis-sentinel /usr/local/redis/conf/sentinel.conf

4、日志文件

|  |
| --- |
| oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo  Redis version=6.0.14, bits=64, commit=00000000, modified=0, pid=1809, just started  #加载配置  Configuration loaded  Increased maximum number of open files to 10032 (it was originally set to 1024).  Could not create server TCP listening socket 0.0.0.0:6379: bind: Address already in use  oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo  Redis version=6.0.14, bits=64, commit=00000000, modified=0, pid=1815, just started  Configuration loaded  Increased maximum number of open files to 10032 (it was originally set to 1024).  #运行sentinel  Running mode=sentinel, port=26379.  WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.#  #创建Sentinel ID  Sentinel ID is de740a49c18fd0ceca774fa6c1ca9f037ec09a4e  #主节点,仲裁的值2  +monitor master mymaster 192.168.205.129 6379 quorum 2  #从节点加到环境中  +slave slave 192.168.205.128:6379 192.168.205.128 6379 @ mymaster 192.168.205.129 6379  +slave slave 192.168.205.130:6379 192.168.205.130 6379 @ mymaster 192.168.205.129 6379  +sentinel sentinel cdc3fd16f6a094611c689e6c3618e57f922552df 192.168.205.130 26379 @ mymaster 192.168.205.129 6379  +sentinel sentinel 00b83e4279b3b3397a18cca2400ad6562eb34a42 192.168.205.128 26379 @ mymaster 192.168.205.129 6379 |

### 哨兵的工作原理

Sentinel内部有3个定时任务,分别是:

每1秒每个Sentinel对其他Sentinel和Redis节点执行PING操作(监控)

每2秒每个Sentinel通过Master节点的channel交换信息(Publish/Subscribe)

每10秒每个Sentinel会对Master和Slave执行INFO命令(完整的redis节点和状态信息。)

主观下线

所谓主观下线(subjectively Down, 简称SDOWN)指的是单个Sentinel实例对服务器做出的下线判断,即单个Sentinel认为某个服务下线(有可能是接收不到订阅,之间的网络不通等等原因)。

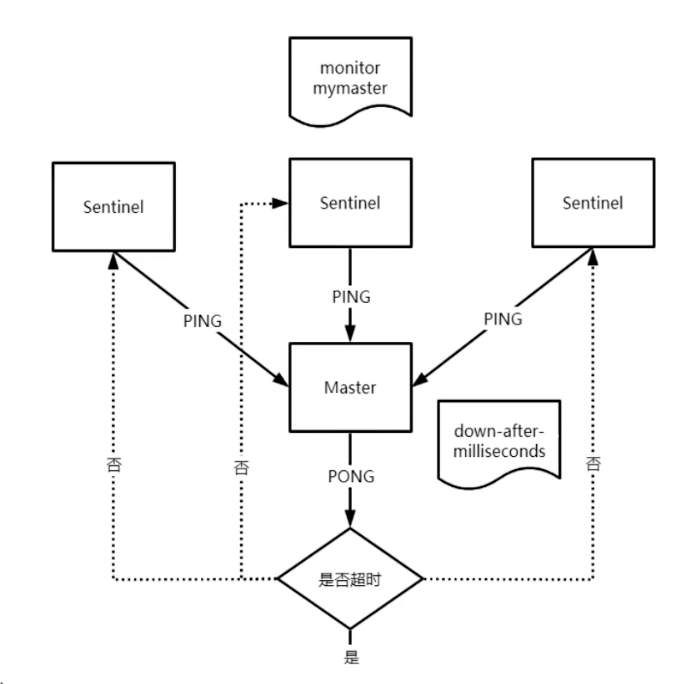
客观下线：

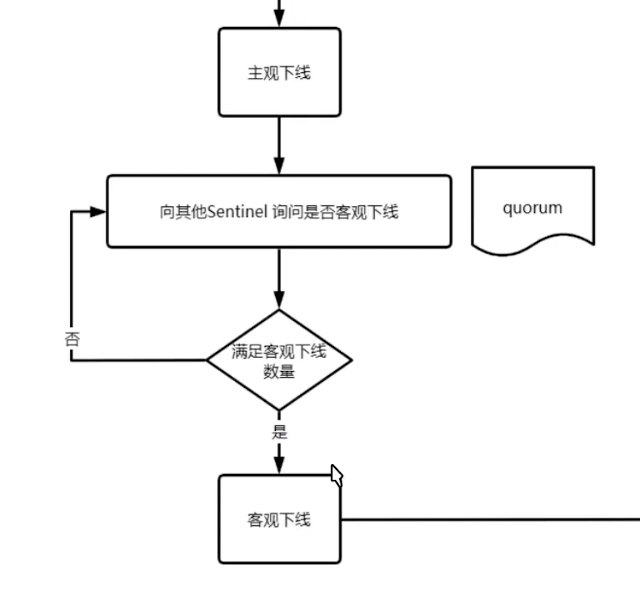
客观下线(objectively Down, 简称QDOWN)指的是多个Sentinel实例在对同一个服务器做出SDOWN判断,并且通过命令互相交流之后,得出的服务器下线判断,然后开启failover（故障迁移）.

仲裁:

仲裁指的是配置文件中的quorum选项。quorum的值一般设置为Septinel个数的二分之一加1,例如3个Sentinel就设置为2.

### 哨兵的工作流程





Monitor mymaster:sentienl 监控的主节点的地址。

Down-fater-millisenonds:满足故障迁移的时间。

1、sentinel每秒PING master做心跳检测。

sentinel每秒ping 每个sentinel做心跳检测。

sentinel每2秒与master交换信息(发布订阅)。

entinel每10秒与master做info信息交换。

2、有效回复PING命令的时间超过配置文件down-after-milliseconds选项所指定的值,被认定为主观下线

3、确认主观下线状态

4、满足条件,客观下线

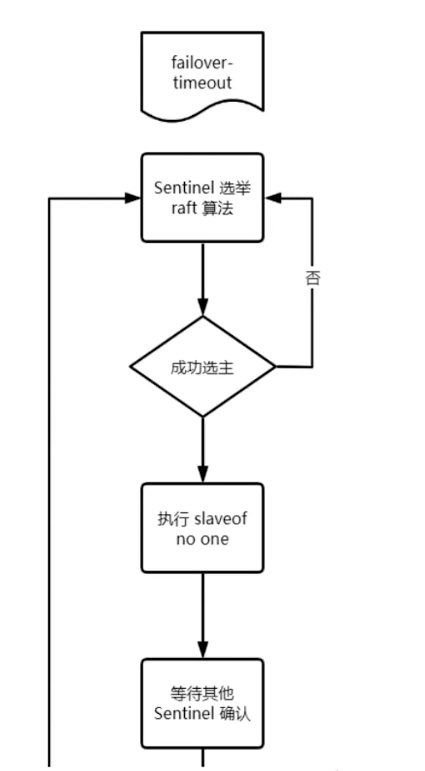
5、投票选举主机点,从节点复制数据

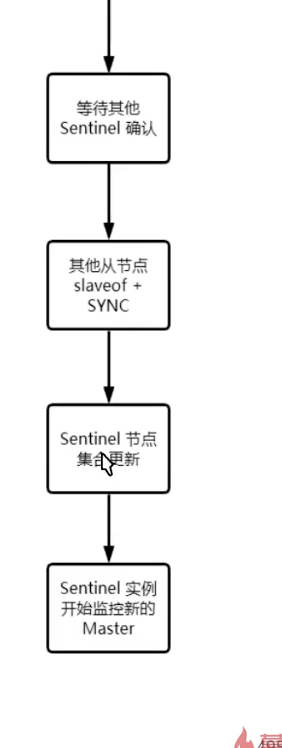
6、当主节点标记为客观下线时, INFO命令触发由10s一次改为1s一次。

7、若没有足够数量的Sentinel同意Master已经下线, Master的客观下线状态就会被移除。若Master重新向Sentinel的PING命令返回有效回复, Master的主观下线状态就会被移除。

故障迁移

Faliover-timeout:故障迁移超时时间。





### 哨兵故障演示

1、启动服务

启动三个redis

/usr/local/redis/bin/redis-server /usr/local/redis/conf/redis.conf

启动三个Sentinel

/usr/local/redis/bin/redis-sentinel /usr/local/redis/conf/sentinel.conf

2、关闭主结点

Kill pid

|  |
| --- |
| #当前sentinel标记该结点为主观下线  +sdown master mymaster 192.168.205.129 6379  #所有的sentinel满足客观下线条件  +odown master mymaster 192.168.205.129 6379 #quorum 3/2  #开启新的纪元,启动新的选举流程，并记录选举次数。  +new-epoch 1  #开启故障迁移，主节点。重新选举主节点,并做数据同步。  +try-failover master mymaster 192.168.205.129 6379  #投票，选一个leader去做故障转移工作。该节点现在为当前机器  +vote-for-leader cdc3fd16f6a094611c689e6c3618e57f922552df 1  de740a49c18fd0ceca774fa6c1ca9f037ec09a4e voted for cdc3fd16f6a094611c689e6c3618e57f922552df 1  00b83e4279b3b3397a18cca2400ad6562eb34a42 voted for cdc3fd16f6a094611c689e6c3618e57f922552df 1  #对谁做故障转移。  +elected-leader master mymaster 192.168.205.129 6379  #查找主从环境合适的主节点，  1、redis配置中有去配置优先级。  2、看偏移量，与关掉主节点的偏移量。  +failover-state-select-slave master mymaster 192.168.205.129 6379  #找到了合适的结点192.168.205.130作为主节点。  +selected-slave slave 192.168.205.130:6379 192.168.205.130 6379 @ mymaster 192.168.205.129 6379  #关闭当前结点的复制功能。此时当前机器变为朱及诶单  +failover-state-send-slaveof-noone slave 192.168.205.130:6379 192.168.205.130 6379 @ mymaster 192.168.205.129 6379  #等待其他的sentinel 确认升级。  +failover-state-wait-promotion slave 192.168.205.130:6379 192.168.205.130 6379 @ mymaster 192.168.205.129 6379  #确认升级  +promoted-slave slave 192.168.205.130:6379 192.168.205.130 6379 @ mymaster 192.168.205.129 6379  #确认完毕  +failover-state-reconf-slaves master mymaster 192.168.205.129 6379  #对其他的结点修改配置。  +slave-reconf-sent slave 192.168.205.128:6379 192.168.205.128 6379 @ mymaster 192.168.205.129 6379  #修改成功  +slave-reconf-inprog slave 192.168.205.128:6379 192.168.205.128 6379 @ mymaster 192.168.205.129 6379  #节点客观下线,  -odown master mymaster 192.168.205.129 6379  #开始都进行sync复制。  +slave-reconf-done slave 192.168.205.128:6379 192.168.205.128 6379 @ mymaster 192.168.205.129 6379  #故障迁移结束。  +failover-end master mymaster 192.168.205.129 6379  #新的结点的变更。  +switch-master mymaster 192.168.205.129 6379 192.168.205.130 6379  #新的主从关系。  +slave slave 192.168.205.128:6379 192.168.205.128 6379 @ mymaster 192.168.205.130 6379  +slave slave 192.168.205.129:6379 192.168.205.129 6379 @ mymaster 192.168.205.130 6379  +sdown slave 192.168.205.129:6379 192.168.205.129 6379 @ mymaster 192.168.205.130 6379  +tilt #tilt mode entered  -tilt #tilt mode exited |

3、nfo 当前机器，当前结点的信息变为主节点 master。

4、重新启动挂掉的机器，会重新加入到监控中。

原来master配置文件修改为

|  |
| --- |
| user default on #8d969eef6ecad3c29a3a629280e686cf0c3f5d5a86aff3ca12020c923adc6c92 ~\* +@all  #要复制的新的结点。  replicaof 192.168.205.130 6379 |

升级为master的结点

|  |
| --- |
| user default on #8d969eef6ecad3c29a3a629280e686cf0c3f5d5a86aff3ca12020c923adc6c92 ~\* +@all  #从节点属于那个主节点 这个配置没有了 |

### 故障迁移一致性

多个哨兵都要去选择主节点？

这里用到了分布式一致性的算法Raft共识算法,就是怎样选举Sentinel节点为领头节点。

1) Sentinel自动故障迁移使用Raf算法来选举领头(leader) Sentinel ,从而确保在一个给定的周期(epoch)里,只有一个领头产生。

2)这表示在同一个周期中,不会有两个Sentinel同时被选中为领头,并且各个Sentinel在同一个节点中只会对一个领头进行投票。

3)更高的配置节点总是优于较低的节点,因此每个Sentinel都会主动使用更新的节点来代替自己的配置。

简单来说,我们可以将Sentinel配置看作是一个带有版本号的状态。一个状态会以最后写入者胜出(last-write-wins)的方式(也

即是,最新的配置总是胜出)传播至所有其他Sentinel.

### TITL模式



# Redis集群

## Redis 集群

### 为什么需要Cluster

### 引入

单机:单机故障、读写压力、存储数据较少。

主从: 解决单机故障，读写的分离。主节点和从节点的切换。

哨兵:主节点和从节点的切换。

集群:redis的数据量比较大，主节点写压力大。

1、数据量

主从和哨兵结构都是单master,master和slave的数据是一样的,所以系统的数据存储量就是redis的内存大小，无法进行redis的横向扩容。

2、写压力

主从和哨兵结构写的压力都放到的master上。

### 集群需要解决的问题

读写分离:读请求请求放到slave，写请求放到master。

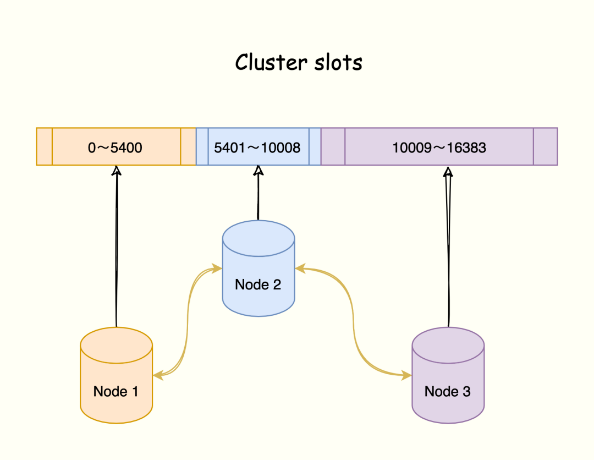
高可用:主节点挂了，之后可以使用从节点去操作。

### Redis集群

Redis 集群是一种分布式数据库方案，集群通过分片（sharding）来进行数据管理（「分治思想」的一种实践），并提供复制和故障转移功能。

将数据划分为 16384 的 slots，每个节点负责一部分槽位。槽位的信息存储于每个节点中。

它是去中心化的，如图所示，该集群有三个 Redis 节点组成，每个节点负责整个集群的一部分数据，每个节点负责的数据多少可能不一样。

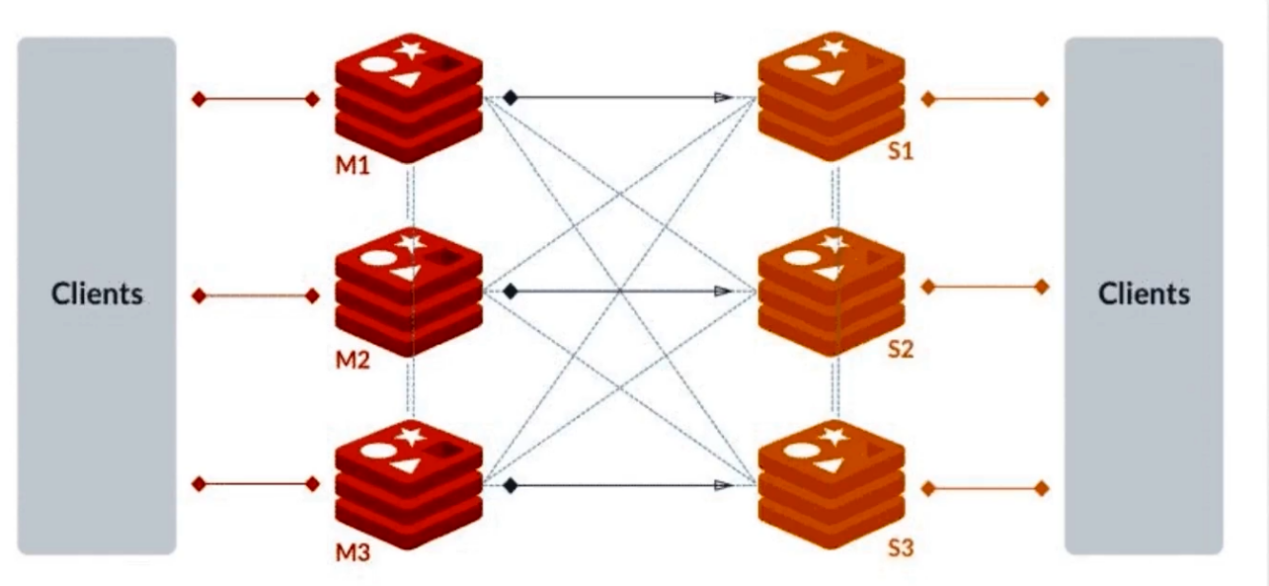


三个节点相互连接组成一个对等的集群，它们之间通过 Gossip协议相互交互集群信息，最后每个节点都保存着其他节点的 slots 分配情况。

### Redis 集群架构

并发量大了一>主从复制解决一>主从稳定性一哨兵解决一>单节点存储能力->集群Cluster解决

Redis cluster集群模式具有高可用、可扩展性、分布式、容错等特性



Redis cluster采用无中心结构,每个节点都可以保存数据和整个集群状态,每个节点都和其他所有节点连接。Cluster至少为6个才能保证组成完整高可用的集群,其中三主三从。主节点分配槽,处理客户端的命令请求,从节点可用在主节点故障后，顶替主节点。

因为最小的Redis集群,需要至少3个主节点,既然有3个主节点,而一个主节点搭配至少一个从节点,因此至少得6台Redis。如上图所示,该集群中包含6个Redis节点, 3主3从,分别为M1, M2, M3, S1, S2, S3,除了主从Redis节点之间进行数据复制外,所有Redis节点之间采用Gossip协议进行通信,交换维护节点元数据信息.

总结下来就是:读请求分配给Slave节点,写请求分配给Master,数据同步从Master到Slave节点。

### 优缺点

优点：

去中心化

可扩展性

高可用性

自动故障转移

缺点：

数据通过异步复制,无法保证数据强一致性;

集群环境搭建略微复杂。

### 数据分区

随着请求量和数据量的增加,一台机器已经无法满足需求,我们就需要把数据和请求分散到多台机器。这时候就需要引入分布

式存储。分布式存储有以下特性:

1.增强可用性

2.维护方便

3.均衡1/0

4.改善查询性能

分布式存储首先要解决把整个数据集按照分区规则映射到多个节点的问题,即把数据集划分到多个节点上,每个节点负责整体数据的一个子集。

## 常见分区算法

### 1、范围分区

优点:同一范围内的范围查询不需要跨节点,提升查询速度

应用场景: MySQL, Oracle

### 2、hash分区

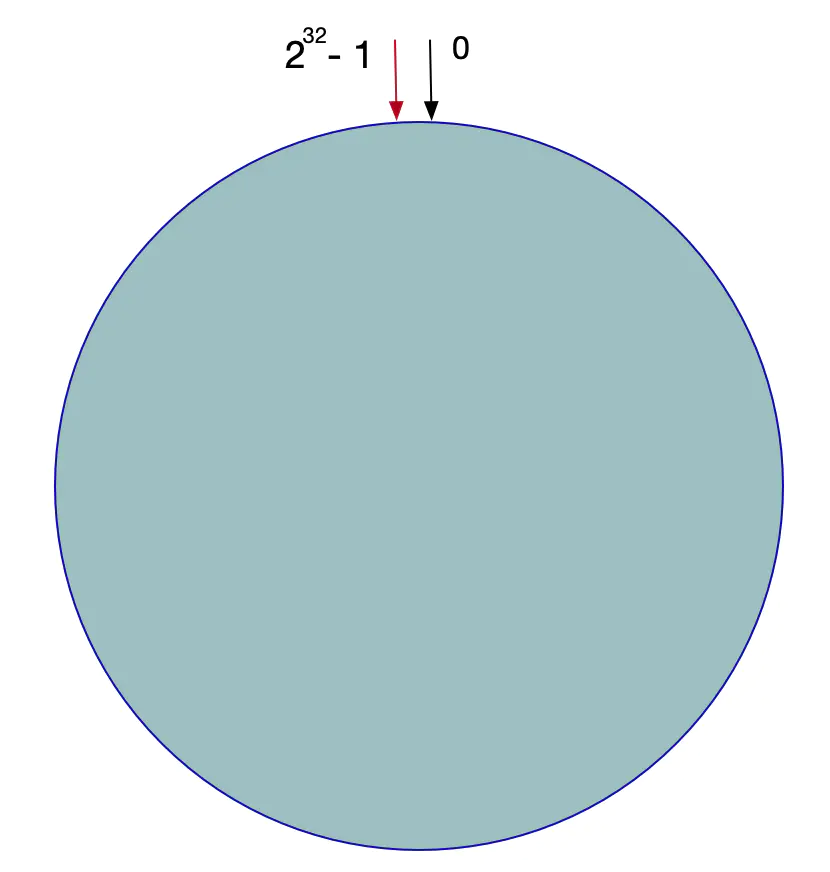
结点取余分区:hash(object)%N

优点:实现简单

缺点: 当扩容或收缩节点时,需要迁移的数据量大。(翻倍扩容可以相对减少迁移量)

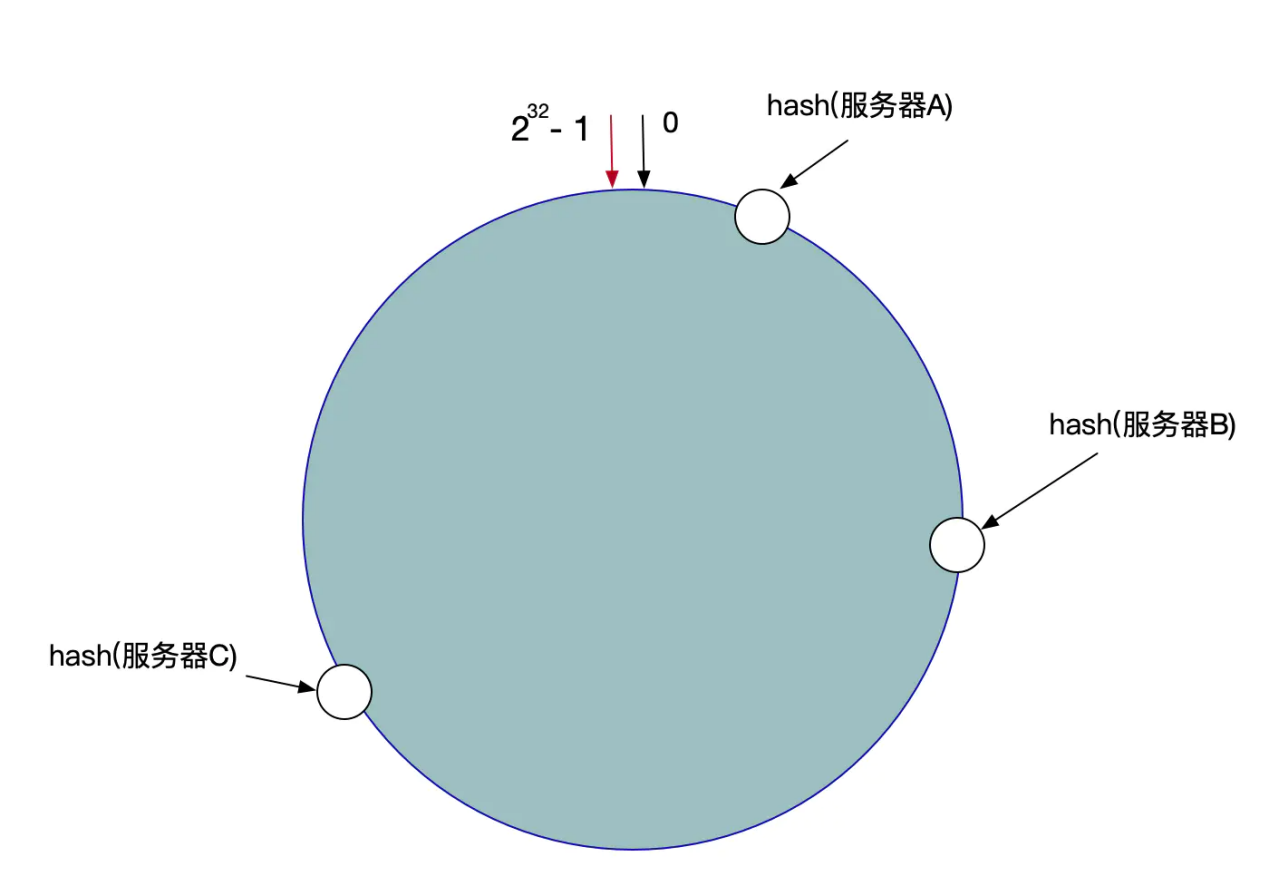
### 一致性hash分区:

一致性Hash算法也是使用取模的方法，不过，上述的取模方法是对服务器的数量进行取模，而一致性的Hash算法是对2的32方取模。即，一致性Hash算法将整个Hash空间组织成一个虚拟的圆环，Hash函数的值空间为0 ~ 2^32 - 1(一个32位无符号整型)，整个哈希环如下：



整个圆环以顺时针方向组织，圆环正上方的点代表0，0点右侧的第一个点代表1，以此类推。

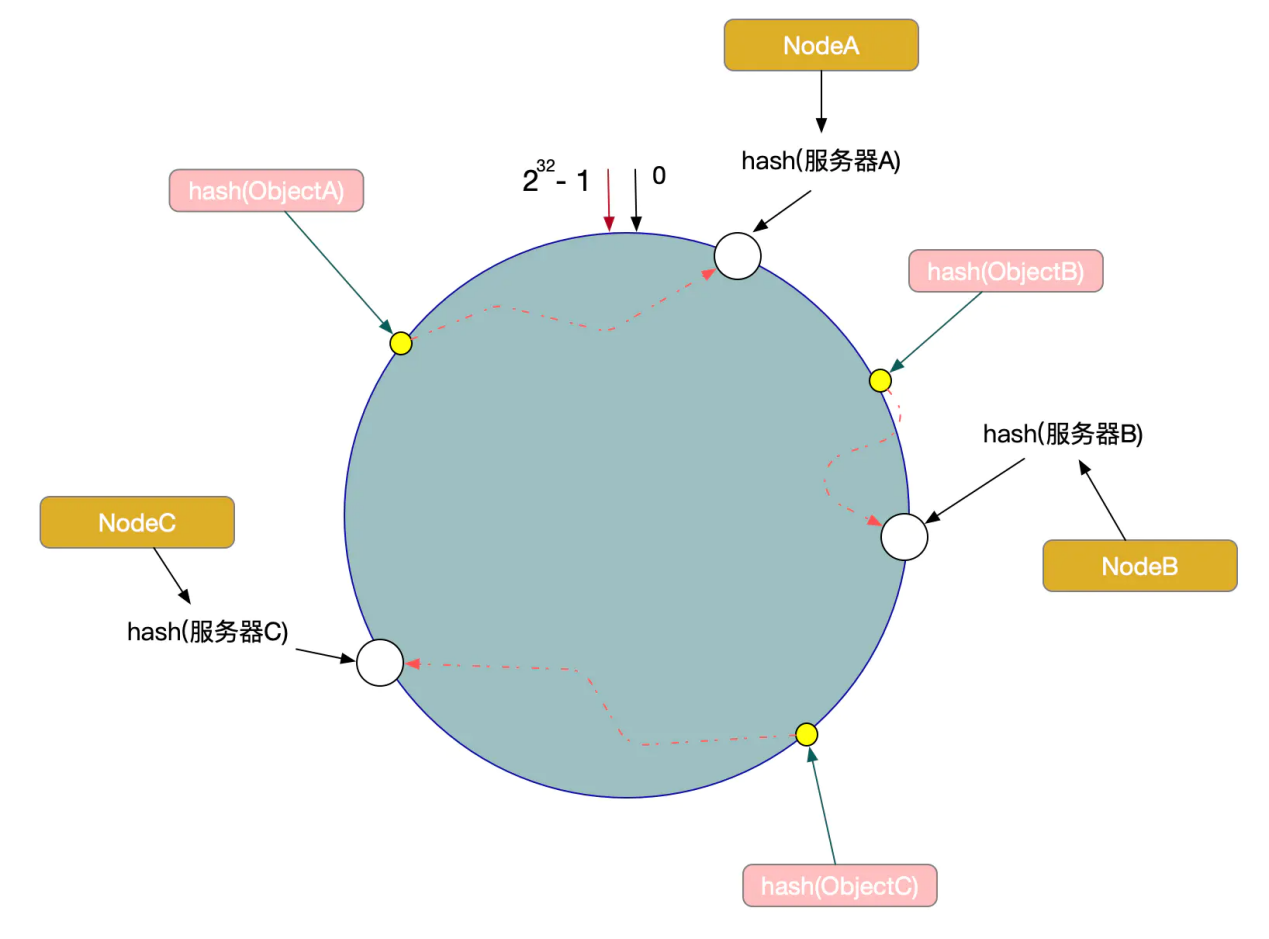
第二步，我们将各个服务器使用Hash进行一个哈希，具体可以选择服务器的IP或主机名作为关键字进行哈希，这样每台服务器就确定在了哈希环的一个位置上，比如我们有三台机器，使用IP地址哈希后在环空间的位置如图1-4所示：



现在，我们使用以下算法定位数据访问到相应的服务器：

将数据Key使用相同的函数Hash计算出哈希值，并确定此数据在环上的位置，从此位置沿环顺时针查找，遇到的服务器就是其应该定位到的服务器。

例如，现在有ObjectA，ObjectB，ObjectC三个数据对象，经过哈希计算后，在环空间上的位置如下：

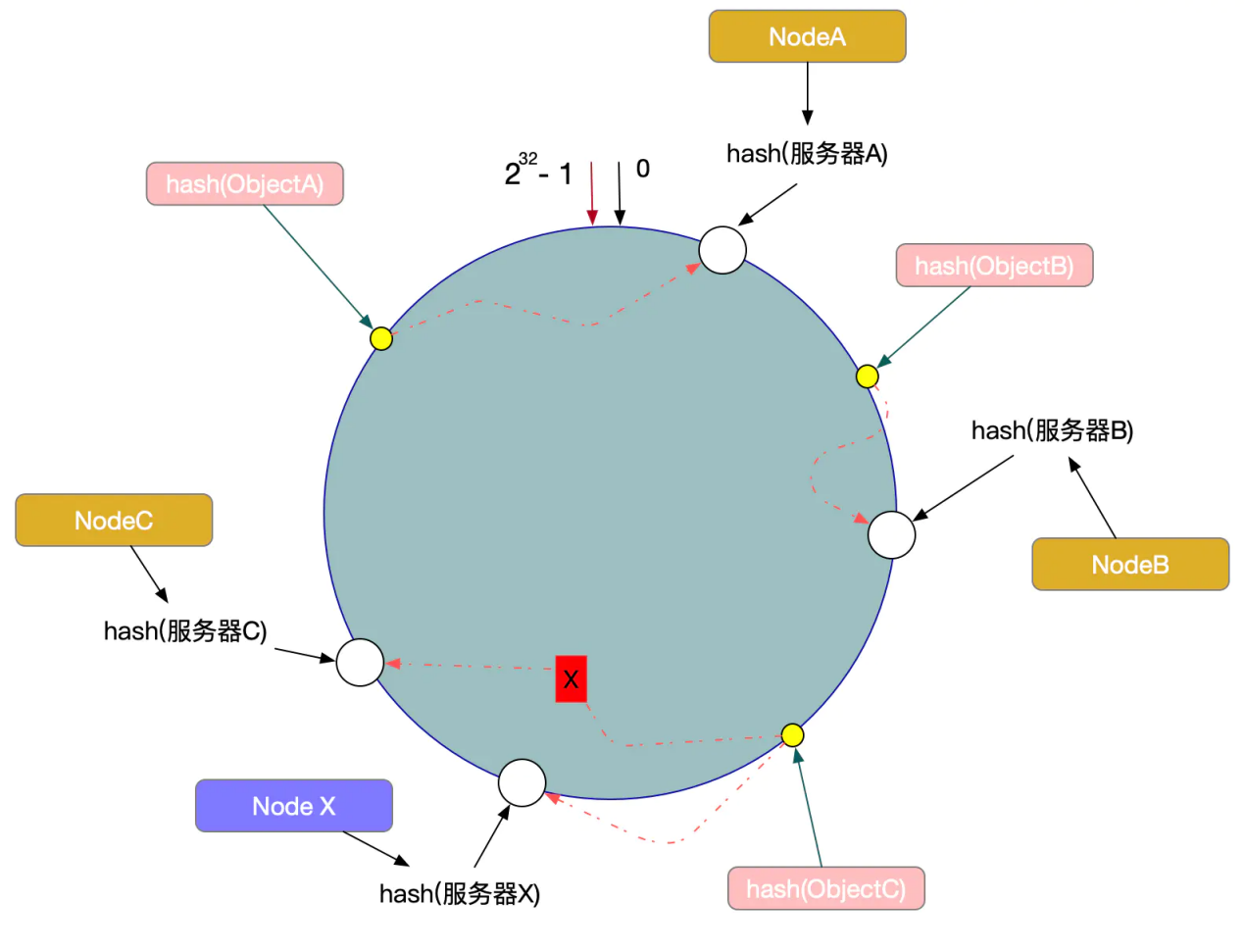


根据一致性算法，Object -> NodeA，ObjectB -> NodeB, ObjectC -> NodeC

一致性Hash算法的容错性和可扩展性

现在，假设我们的Node C宕机了，我们从图中可以看到，A、B不会受到影响，只有Object C对象被重新定位到Node A。所以我们发现，在一致性Hash算法中，如果一台服务器不可用，受影响的数据仅仅是此服务器到其环空间前一台服务器之间的数据（这里为Node C到Node B之间的数据），其他不会受到影响。如图1-6所示：

另外一种情况，现在我们系统增加了一台服务器Node X，如图1-7所示：



一致性Hash算法对于节点的增减都只需重定位环空间中的一小部分数据，有很好的容错性和可扩展性。

优点:相比节点取余最大的好处在于加入和删除节点只影响哈希环中相邻的节点,对其他节点无影响。

缺点:当使用少量节点时,节点变化将大范围影响哈希环中数据映射,因此这种方式不适合少量数据节点的分布式方案。

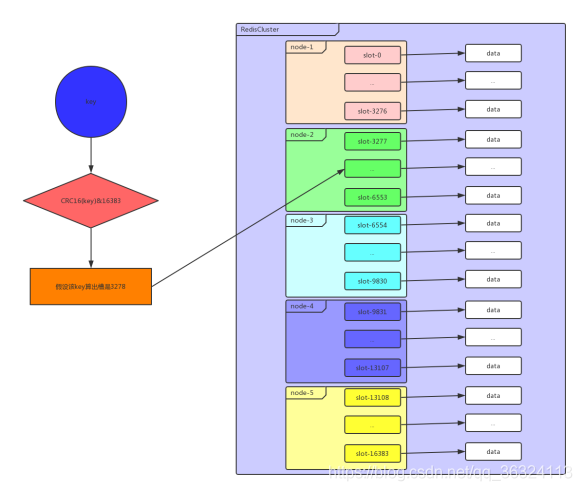
### 虚拟槽分区



虚拟槽分区巧妙地使用了哈希空间，使用分散度良好的哈希函数把所有数据映射到一个固定范围的整数集合中，整数定义为槽（slot）。这个范围一般远远大于节点数，比如 Redis Cluster 槽范围是0~16383。槽是集群内数据管理和迁移的基本单位。采用大范围槽的主要目的是为了方便数据拆分和集群扩展。每个节点会负责一定数量的槽，下图所示。

当前集群有5个节点，每个节点平均大约负责3276个槽。由于采用高质量的哈希算法，每个槽所映射的数据通常比较均匀，将数据平均划分到5个节点进行数据分区。Redis Cluster 就是采用虚拟槽分区，下面就介绍 Redis 数据分区方法。

每当 key 访问过来，Redis Cluster 会计算哈希值是否在这个区间里。它们彼此都知道对应的槽在哪台机器上，这样就能做到平均分配了。



优点:每个node均匀的分配了slot,缩小增减节点影响的范围

缺点:需要存储node和slot的对应信息

## Redis集群原理

Redis 3.0 开始，官方提供了 Redis Cluster 方案实现了切片集群，该方案就实现了数据和实例的规则。Redis Cluster 方案采用哈希槽（Hash Slot，接下来我会直接称之为 Slot），来处理数据和实例之间的映射关系。

### 分片

集群的整个数据库被分为 16384 个槽（slot），数据库中的每个键都属于这 16384 个槽的其中一个，集群中的每个节点可以处理 0 个或最多 16384 个槽。

Key 与哈希槽映射过程可以分为两大步骤：

1、根据键值对的 key，使用 CRC16 算法，计算出一个 16 bit 的值；

2、将 16 bit 的值对 16384 执行取模，得到 0 ～ 16383 的数表示 key 对应的哈希槽。

Cluster 还允许用户强制某个 key 挂在特定槽位上，通过在 key 字符串里面嵌入 tag 标记，这就可以强制 key 所挂在的槽位等于 tag 所在的槽位。

### 哈希槽与 Redis 实例映射

在 部署集群的样例中通过 cluster create 创建，Redis 会自动将 16384 个 哈希槽平均分布在集群实例上，比如 N 个节点，每个节点上的哈希槽数 = 16384 / N 个。

除此之外，可以通过 CLUSTER MEET 命令将 7000、7001、7002 三个节点连在一个集群，但是集群目前依然处于下线状态，因为三个实例都没有处理任何哈希槽。

可以使用 cluster addslots 命令，指定每个实例上的哈希槽个数。

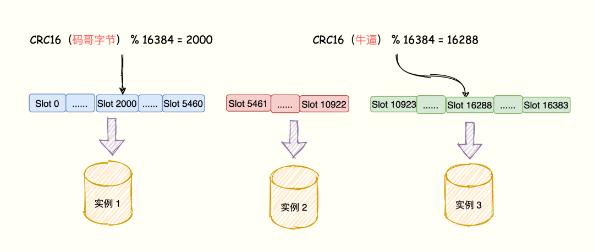
三个实例的集群，通过下面的指令为每个实例分配哈希槽：实例 1负责 0 ～ 5460 哈希槽，实例 2 负责 5461~10922 哈希槽，实例 3 负责 10923 ～ 16383 哈希槽

redis-cli -h 172.16.19.1 –p 6379 cluster addslots 0,5460

redis-cli -h 172.16.19.2 –p 6379 cluster addslots 5461,10922

redis-cli -h 172.16.19.3 –p 6379 cluster addslots 10923,16383

键值对数据、哈希槽、Redis 实例之间的映射关系如下：



Redis 键值对的 key 「码哥字节」「牛逼」经过 CRC16 计算后再对哈希槽总个数 16394 取模，模数结果分别映射到实例 1 与实例 2 上。

切记，当 16384 个槽都分配完全，Redis 集群才能正常工作。

### 复制和故障转移

Master 用于处理槽，Slave 节点则通过《Redis 主从架构数据同步》方式同步主节点数据。

当 Master 下线，Slave 代替主节点继续处理请求。主从节点之间并没有读写分离， Slave 只用作 Master 宕机的高可用备份。

Redis Cluster 可以为每个主节点设置若干个从节点，单主节点故障时，集群会自动将其中某个从节点提升为主节点。

如果某个主节点没有从节点，那么当它发生故障时，集群将完全处于不可用状态。

不过 Redis 也提供了一个参数cluster-require-full-coverage可以允许部分节点故障，其它节点还可以继续提供对外访问。

比如 7000 主节点宕机，作为 slave 的 7003 成为 Master 节点继续提供服务。当下线的节点 7000 重新上线，它将成为当前 70003 的从节点。

### 故障检测

一个节点认为某个节点失联了并不代表所有的节点都认为它失联了。只有当大多数负责处理 slot 节点都认定了某个节点下线了，集群才认为该节点需要进行主从切换。

Redis 集群节点采用 Gossip 协议来广播自己的状态以及自己对整个集群认知的改变。比如一个节点发现某个节点失联了 (PFail)，它会将这条信息向整个集群广播，其它节点也就可以收到这点失联信息。

如果一个节点收到了某个节点失联的数量 (PFail Count) 已经达到了集群的大多数，就可以标记该节点为确定下线状态 (Fail)，然后向整个集群广播，强迫其它节点也接收该节点已经下线的事实，并立即对该失联节点进行主从切换。

### 故障转移

当一个 Slave 发现自己的主节点进入已下线状态后，从节点将开始对下线的主节点进行故障转移。

1、从下线的 Master 及节点的 Slave 节点列表选择一个节点成为新主节点。

2、新主节点会撤销所有对已下线主节点的 slot 指派，并将这些 slots 指派给自己。

3、新的主节点向集群广播一条 PONG 消息，这条 PONG 消息可以让集群中的其他节点立即知道这个节点已经由从节点变成了主节点，并且这个主节点已经接管了原本由已下线节点负责处理的槽。

4、新的主节点开始接收处理槽有关的命令请求，故障转移完成。

### 选主流程

1、集群的配置纪元 +1，是一个自曾计数器，初始值 0 ，每次执行故障转移都会 +1。

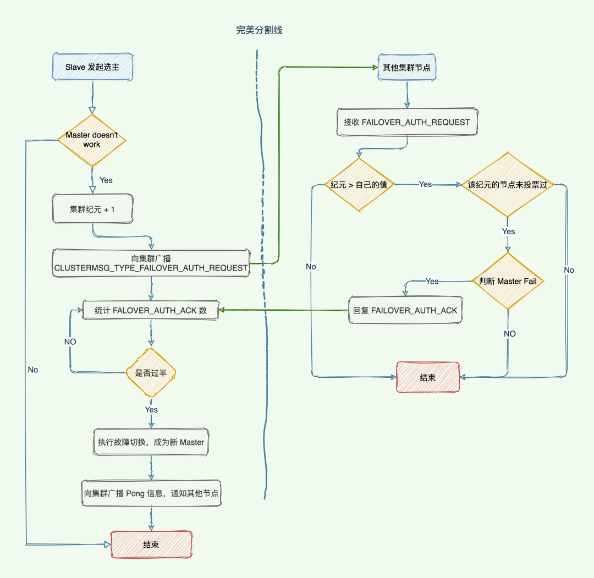
2、检测到主节点下线的从节点向集群广播一条CLUSTERMSG\_TYPE\_FAILOVER\_AUTH\_REQUEST消息，要求所有收到这条消息、并且具有投票权的主节点向这个从节点投票。

3、这个主节点尚未投票给其他从节点，那么主节点将向要求投票的从节点返回一条CLUSTERMSG\_TYPE\_FAILOVER\_AUTH\_ACK消息，表示这个主节点支持从节点成为新的主节点。

4、参与选举的从节点都会接收CLUSTERMSG\_TYPE\_FAILOVER\_AUTH\_ACK消息，如果收集到的票 >= (N/2) + 1 支持，那么这个从节点就被选举为新主节点。

5、如果在一个配置纪元里面没有从节点能收集到足够多的支持票，那么集群进入一个新的配置纪元，并再次进行选举，直到选出新的主节点为止。

跟哨兵类似，两者都是基于 Raft 算法来实现的，流程如图所示



### 客户端定位实例

Redis 实例会将自己的哈希槽信息通过 Gossip 协议发送给集群中其他的实例，实现了哈希槽分配信息的扩散。

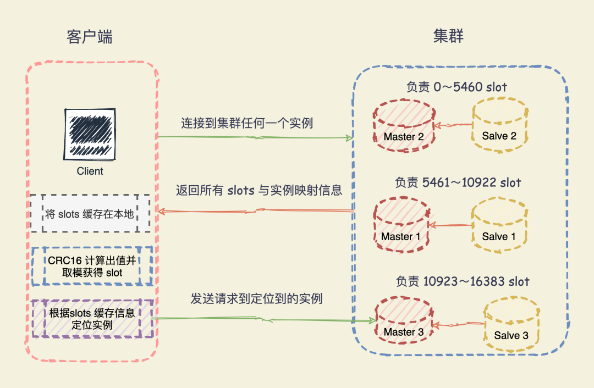
这样，集群中的每个实例都有所有哈希槽与实例之间的映射关系信息。

在切片数据的时候是将 key 通过 CRC16 计算出一个值再对 16384 取模得到对应的 Slot，这个计算任务可以在客户端上执行发送请求的时候执行。

但是，定位到槽以后还需要进一步定位到该 Slot 所在 Redis 实例。

当客户端连接任何一个实例，实例就将哈希槽与实例的映射关系响应给客户端，客户端就会将哈希槽与实例映射信息缓存在本地。

当客户端请求时，会计算出键所对应的哈希槽，在通过本地缓存的哈希槽实例映射信息定位到数据所在实例上，再将请求发送给对应的实例。



### 重新分配哈希槽

集群中的实例通过 Gossip 协议互相传递消息获取最新的哈希槽分配信息，但是，客户端无法感知。

Redis Cluster 提供了重定向机制：客户端将请求发送到实例上，这个实例没有相应的数据，该 Redis 实例会告诉客户端将请求发送到其他的实例上。

Redis 如何告知客户端重定向访问新实例呢

分为两种情况：MOVED 错误、ASK 错误。

**MOVED 错误**

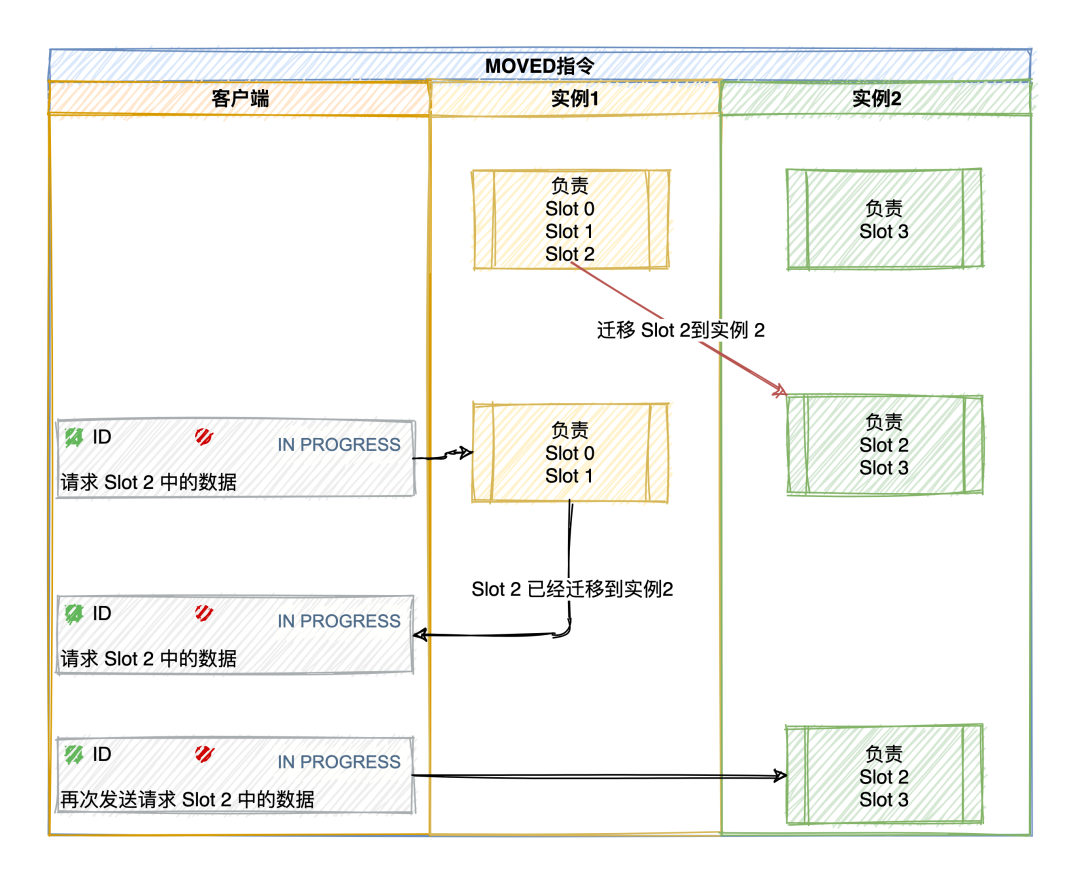
MOVED 错误（负载均衡，数据已经迁移到其他实例上）：当客户端将一个键值对操作请求发送给某个实例，而这个键所在的槽并非由自己负责的时候，该实例会返回一个 MOVED 错误指引转向正在负责该槽的节点。

GET 码哥字节

(error) MOVED 16330 172.17.18.2:6379

该响应表示客户端请求的键值对所在的哈希槽 16330 迁移到了 172.17.18.2 这个实例上，端口是 6379。这样客户端就与 172.17.18.2:6379 建立连接，并发送 GET 请求。

同时，客户端还会更新本地缓存，将该 slot 与 Redis 实例对应关系更新正确。



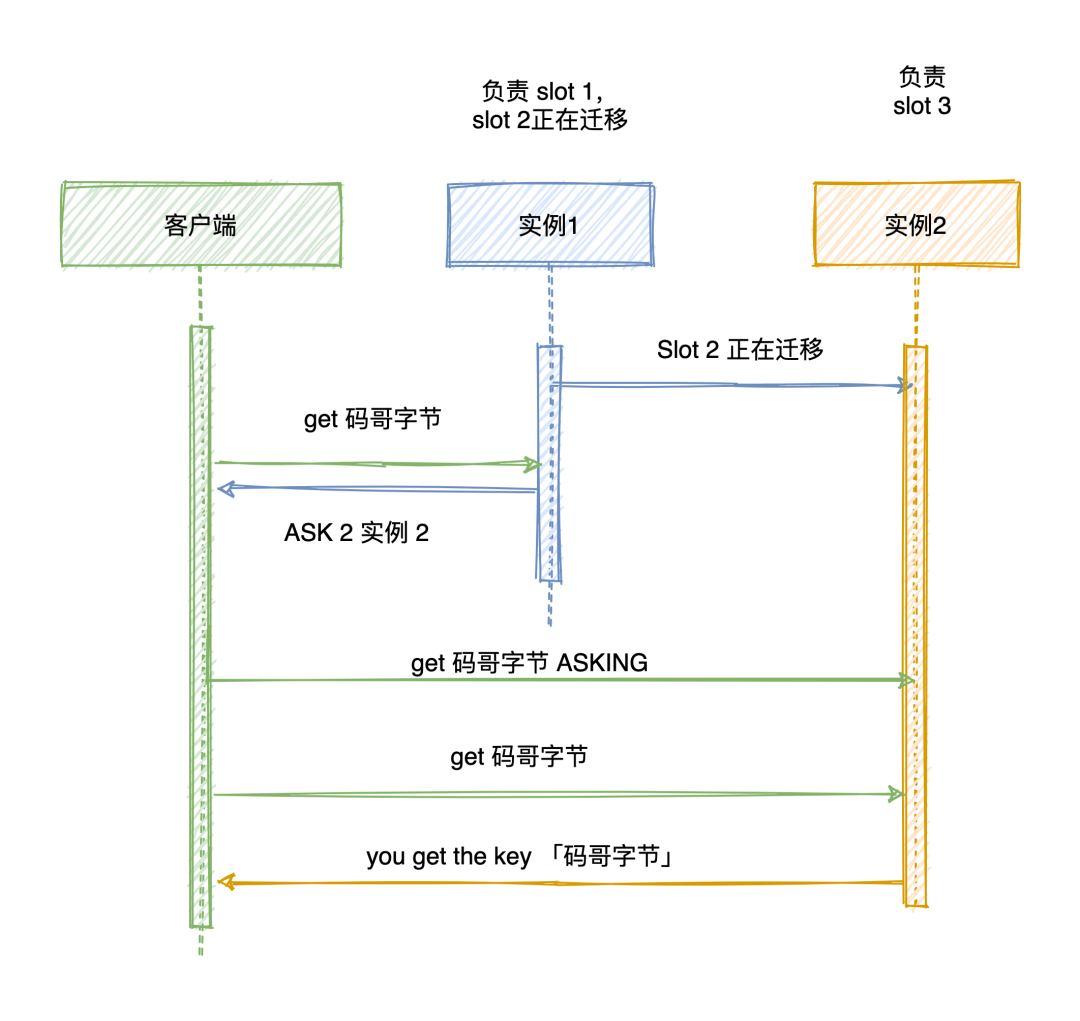
**ASK 错误**  
 如果某个 slot 的数据比较多，部分迁移到新实例，还有一部分没有迁移咋办？

如果请求的 key 在当前节点找到就直接执行命令，否则时候就需要 ASK 错误响应了，槽部分迁移未完成的情况下，如果需要访问的 key 所在 Slot 正在从从 实例 1 迁移到 实例 2，实例 1 会返回客户端一条 ASK 报错信息：客户端请求的 key 所在的哈希槽正在迁移到实例 2 上，你先给实例 2 发送一个 ASKING 命令，接着发发送操作命令。

GET 码哥字节

(error) ASK 16330 172.17.18.2:6379

比如客户端请求定位到 key = 「码哥字节」的槽 16330 在实例 172.17.18.1 上，节点 1 如果找得到就直接执行命令，否则响应 ASK 错误信息，并指引客户端转向正在迁移的目标节点 172.17.18.2。

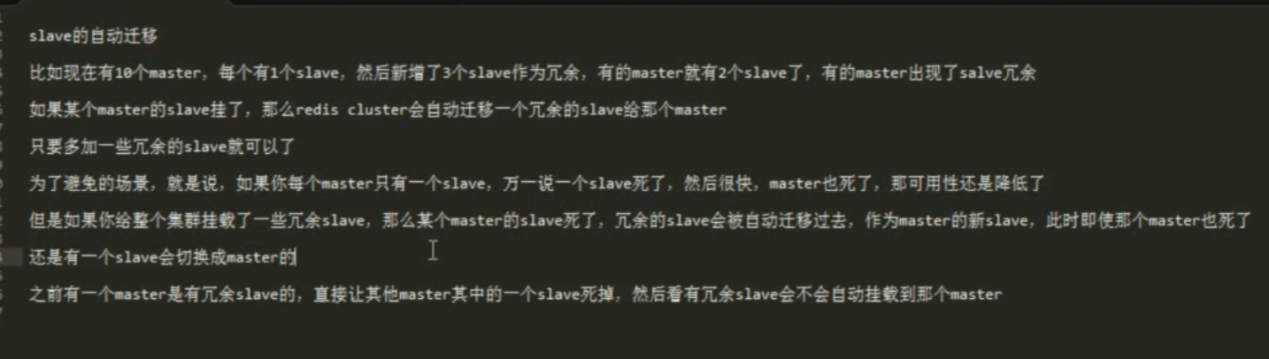


注意：ASK 错误指令并不会更新客户端缓存的哈希槽分配信息。

所以客户端再次请求 Slot 16330 的数据，还是会先给 172.17.18.1 实例发送请求，只不过节点会响应 ASK 命令让客户端给新实例发送一次请求。

MOVED指令则更新客户端本地缓存，让后续指令都发往新实例。

### 自动化salve迁移



## 结点通信

### Gossip 消息

发送的消息结构是 clusterMsgDataGossip结构体组成：

|  |
| --- |
| typedef struct {  char nodename[CLUSTER\_NAMELEN]; //40字节  uint32\_t ping\_sent; //4字节  uint32\_t pong\_received; //4字节  char ip[NET\_IP\_STR\_LEN]; //46字节  uint16\_t port; //2字节  uint16\_t cport; //2字节  uint16\_t flags; //2字节  uint32\_t notused1; //4字节  } clusterMsgDataGossip |

所以每个实例发送一个 Gossip消息，就需要发送 104 字节。如果集群是 1000 个实例，那么每个实例发送一个 PING 消息则会占用 大约 10KB。

除此之外，实例间在传播 Slot 映射表的时候，每个消息还包含了 一个长度为 16384 bit 的 Bitmap。

每一位对应一个 Slot，如果值 = 1 则表示这个 Slot 属于当前实例，这个 Bitmap 占用 2KB，所以一个 PING 消息大约 12KB。

PONG与PING 消息一样，一发一回两个消息加起来就是 24 KB。集群规模的增加，心跳消息越来越多就会占据集群的网络通信带宽，降低了集群吞吐量。

### 实例的通信频率

Redis Cluster 的实例启动后，默认会每秒从本地的实例列表中随机选出 5 个实例，再从这 5 个实例中找出一个最久没有收到 PING 消息的实例，把 PING 消息发送给该实例。

Redis Cluster 的实例每 100 ms 就会扫描本地实例列表，当发现有实例最近一次收到 PONG 消息的时间 > cluster-node-timeout / 2。那么就立刻给这个实例发送 PING 消息，更新这个节点的集群状态信息。

当集群规模变大，就会进一步导致实例间网络通信延迟怎加。可能会引起更多的 PING 消息频繁发送。

### 降低实例间的通信开销

每个实例每秒发送一条 PING消息，降低这个频率可能会导致集群每个实例的状态信息无法及时传播。

每 100 ms 检测实例 PONG消息接收是否超过 cluster-node-timeout / 2，这个是 Redis 实例默认的周期性检测任务频率，我们不会轻易修改。

所以，只能修改 cluster-node-timeout的值：集群中判断实例是否故障的心跳时间，默认 15 S。

所以，为了避免过多的心跳消息占用集群宽带，将 cluster-node-timeout调成 20 秒或者 30 秒，这样 PONG 消息接收超时的情况就会缓解。

但是，也不能设置的太大。都则就会导致实例发生故障了，却要等待 cluster-node-timeout时长才能检测出这个故障，影响集群正常服务、

### 到底是什么限制了集群规模呢

关键在于实例间的通信开销，Cluster 集群中的每个实例都保存所有哈希槽与实例对应关系信息（Slot 映射到节点的表），以及自身的状态信息。

在集群之间每个实例通过 Gossip协议传播节点的数据，Gossip 协议工作原理大概如下：

1、从集群中随机选择一些实例按照一定的频率发送 PING 消息发送给挑选出来的实例，用于检测实例状态以及交换彼此的信息。PING 消息中封装了发送者自身的状态信息、部分其他实例的状态信息、Slot 与实例映射表信息。

2、实例接收到 PING 消息后，响应 PONG 消息，消息包含的信息跟 PING 消息一样。

集群之间通过 Gossip协议可以在一段时间之后每个实例都能获取其他所有实例的状态信息。

所以在有新节点加入，节点故障，Slot 映射变更都可以通过 PING，PONG 的消息传播完成集群状态在每个实例的传播同步。

## Redis集群搭建

### Redis集群搭建

端口分配：一条机器挂两个redis

|  |  |
| --- | --- |
| 192.168.205.129 | 6371 |
| 192.168.205.129 | 6372 |
| 192.168.205.128 | 6373 |
| 192.168.205.128 | 6374 |
| 192.168.205.130 | 6375 |
| 192.168.205.130 | 6376 |

1、创建目录

mkdir -p /usr/local/redis/cluster/conf /usr/lcoal/redis/cluster/data /usr/lcoal/redis/cluster/log /usr/lcoal/redis/cluster/conf

2、配置信息。

|  |
| --- |
| #指定Redis只接收来自于该IP地址的请求,多个空格  bind 0.0.0.0  #端口  prot 6376  #默认情况下, redis不是在后台运行的,如果需要在后台运行,把该项的值更改为yes  daemonize yes  #日志文件  logfile "/usr/local/redis/cluster/log/redis-6376.log"  #镜像备份文件的文件名。  dbfilename dump-6376.rdb  #aop持久化开启  appendonly yes  #aop持久化文件名称  appendfilename "appendonly-6376.aof"  #数据库镜像备份的文件放置的路径。  dir /usr/local/redis/cluster/data  #连接密码,没有密码,直接注释  requirepass 123456  #从节点访问主节点密码(必须与requirepass)  masterauth 123456  #是否开启集群模式  cluster-enabled yes  #集群结点信息文件，会保存dir配置对应目录下。  cluster-config-file node-6376.conf  #集群结点链接超时时间  cluster-node-timeout 15000  #集群结点ip  cluster-announce-ip 192.168.205.130  #集群结点映射端口  cluster-announce-prot 6376  #集群结点总线端口  cluster-announce-bus-port 16376 |

3、依次启动

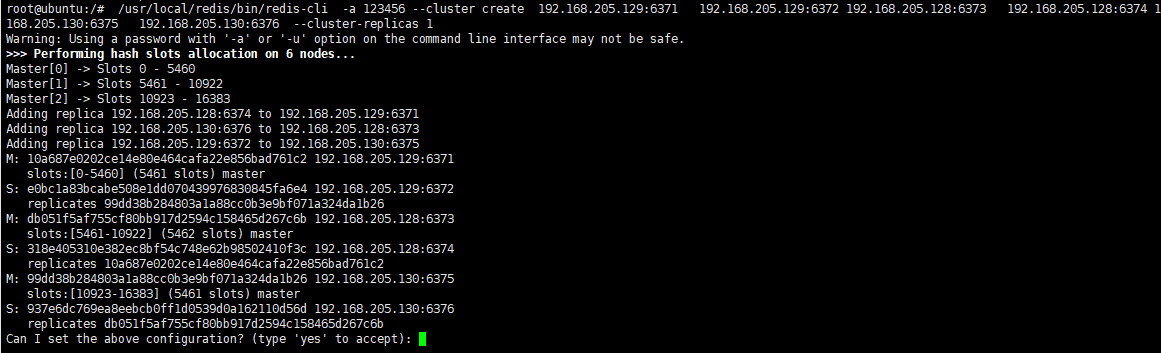
/usr/local/redis/bin/redis-server /usr/local/redis/cluster/conf/redis-6371.conf

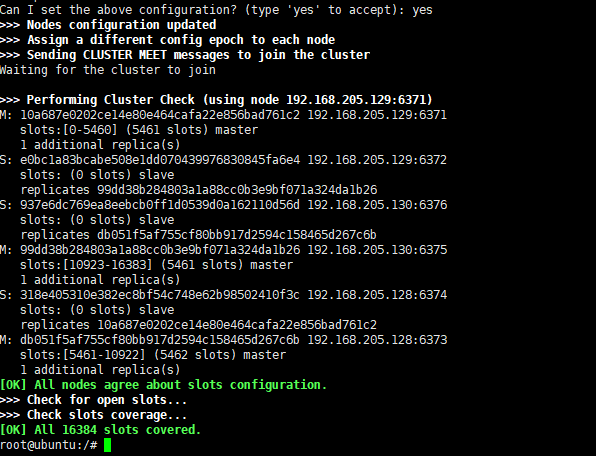
4、创建集群

随便找台机器创建集群 cluster-replicas:主从比例

/usr/local/redis/bin/redis-cli -a 123456 --cluster create 192.168.205.129:6371 192.168.205.129:6372 192.168.205.128:6373 192.168.205.128:6374 192.168.205.130:6375 192.168.205.130:6376 --cluster-replicas 1

Redis分配的槽。





4、集群中常用命令

查看集群信息

/usr/local/redis/bin/redis-cli -a 123456 --cluster check 192.168.205.129:6371

|  |
| --- |
| #主节点信息。  192.168.205.129:6371 (10a687e0...) -> 0 keys | 5461 slots | 1 slaves.  192.168.205.130:6375 (99dd38b2...) -> 0 keys | 5461 slots | 1 slaves.  192.168.205.128:6373 (db051f5a...) -> 0 keys | 5462 slots | 1 slaves.  #多少key在结点中  [OK] 0 keys in 3 masters.  #槽的分配情况。  0.00 keys per slot on average.  #执行check的机器。  >>> Performing Cluster Check (using node 192.168.205.129:6371)  #结点id，结点ip ，结点的槽，从节点数量。复制的结点id。  M: 10a687e0202ce14e80e464cafa22e856bad761c2 192.168.205.129:6371  slots:[0-5460] (5461 slots) master  1 additional replica(s)  S: e0bc1a83bcabe508e1dd070439976830845fa6e4 192.168.205.129:6372  slots: (0 slots) slave  replicates 99dd38b284803a1a88cc0b3e9bf071a324da1b26  S: 937e6dc769ea8eebcb0ff1d0539d0a162110d56d 192.168.205.130:6376  slots: (0 slots) slave  replicates db051f5af755cf80bb917d2594c158465d267c6b  M: 99dd38b284803a1a88cc0b3e9bf071a324da1b26 192.168.205.130:6375  slots:[10923-16383] (5461 slots) master  1 additional replica(s)  S: 318e405310e382ec8bf54c748e62b98502410f3c 192.168.205.128:6374  slots: (0 slots) slave  replicates 10a687e0202ce14e80e464cafa22e856bad761c2  M: db051f5af755cf80bb917d2594c158465d267c6b 192.168.205.128:6373  slots:[5461-10922] (5462 slots) master  1 additional replica(s)  [OK] All nodes agree about slots configuration.  >>> Check for open slots...  >>> Check slots coverage...  [OK] All 16384 slots covered. |

5、主从日志

主节点日志。

|  |
| --- |
| oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo  Redis version=6.0.14, bits=64, commit=00000000, modified=0, pid=4148, just started  Configuration loaded  Increased maximum number of open files to 10032 (it was originally set to 1024).  #没有找到集群配置,自己的id  No cluster configuration found, I'm 10a687e0202ce14e80e464cafa22e856bad761c2  #运行模式 集群模式启动  Running mode=cluster, port=6371.  #linux 相关警告  WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.  #服务初始化。  Server initialized  WARNING overcommit\_memory is set to 0! Background save may fail under low memory condition. To fix this issue add 'vm.overcommit\_memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit\_memory=1' for this to take effect.  #开始接收客户端连接  Ready to accept connections  #给配置中写入纪元时间1  configEpoch set to 1 via CLUSTER SET-CONFIG-EPOCH  #从节点发送sync的请求复制  Replica 192.168.205.128:6374 asks for synchronization  #拒绝部分复制  Partial resynchronization not accepted: Replication ID mismatch (Replica asked for '0cac889b4f1b24008f5c7af11408628e2216a9fd', my replication IDs are '3d78ef666b7f565b46e899d5291e4fc8baee312a' and '0000000000000000000000000000000000000000')  #创建log文件，生成id  Replication backlog created, my new replication IDs are 'd3aacdb5fc4f2582895786dce8960b82d7e11aac' and '0000000000000000000000000000000000000000'  #开始bgsave将数据写到磁盘。  Starting BGSAVE for SYNC with target: disk  #开始子守护进程开始写入  Background saving started by pid 4193  #写入磁盘  DB saved on disk  #0mb写入磁盘  RDB: 0 MB of memory used by copy-on-write  #同步结束。  Background saving terminated with success  Synchronization with replica 192.168.205.128:6374 succeeded  #集群成功  Cluster state changed: ok |

从节点日志:

|  |
| --- |
| oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo  Redis version=6.0.14, bits=64, commit=00000000, modified=0, pid=4582, just started  #配置加载  Configuration loaded  Increased maximum number of open files to 10032 (it was originally set to 1024).  #没有集群配置发现,集群是在链接客户端后创建的。  No cluster configuration found, I'm 318e405310e382ec8bf54c748e62b98502410f3c  #运行集群模式。  Running mode=cluster, port=6374.  WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.  #服务初始化。  Server initialized  WARNING overcommit\_memory is set to 0! Background save may fail under low memory condition. To fix this issue add 'vm.overcommit\_memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit\_memory=1' for this to take effect.  #开始建立链接  Ready to accept connections  configEpoch set to 4 via CLUSTER SET-CONFIG-EPOCH  #变为从节点之前，做一些工作。  Before turning into a replica, using my own master parameters to synthesize a cached master: I may be able to synchronize with the new master with just a partial transfer.  Cluster state changed: ok  #链接主节点  Connecting to MASTER 192.168.205.129:6371  #sync的主从复制  MASTER <-> REPLICA sync started  #触发了sync的非阻塞链接事件。  Non blocking connect for SYNC fired the event.  #master回复了pong,复制继续  Master replied to PING, replication can continue...  #尝试增量复制拒绝。  Trying a partial resynchronization (request 0cac889b4f1b24008f5c7af11408628e2216a9fd:1).  #来自主节点的全量复制。  Full resync from master: d3aacdb5fc4f2582895786dce8960b82d7e11aac:0  #抛弃原来的主节点。  Discarding previously cached master state.  #176字节存到磁盘  MASTER <-> REPLICA sync: receiving 176 bytes from master to disk  #刷新旧的数据。  MASTER <-> REPLICA sync: Flushing old data  #把数据加载到内存。  MASTER <-> REPLICA sync: Loading DB in memory  #RDB相关东西  Loading RDB produced by version 6.0.14  RDB age 0 seconds  RDB memory usage when created 2.53 Mb  #主从复制完成  MASTER <-> REPLICA sync: Finished with success  #后天开启aof  Background append only file rewriting started by pid 4600  #aof 缓存同步。  AOF rewrite child asks to stop sending diffs.  Parent agreed to stop sending diffs. Finalizing AOF...  Concatenating 0.00 MB of AOF diff received from parent.  SYNC append only file rewrite performed  AOF rewrite: 0 MB of memory used by copy-on-write  #后台aof终止。  Background AOF rewrite terminated with success  #aof完成  Residual parent diff successfully flushed to the rewritten AOF (0.00 MB)  Background AOF rewrite finished successfully |

6、查看集群信息

查看命令:/usr/local/redis/bin/redis-cli -c -a 123456 -h 192.168.205.129 -p 6371

CLUSTER info

|  |
| --- |
| #集群状态  cluster\_state:ok  #集群中分配的总槽数  cluster\_slots\_assigned:16384  #成功的  cluster\_slots\_ok:16384  #可能失效的  cluster\_slots\_pfail:0  #已经失效的  cluster\_slots\_fail:0  #环境的结点  cluster\_known\_nodes:6  #集群的数量  cluster\_size:3  #当前环境中纪元的标记。  cluster\_current\_epoch:6  #当前结点的最新的标记。  cluster\_my\_epoch:1  #cluster ping ping 的信息。  cluster\_stats\_messages\_ping\_sent:2095  cluster\_stats\_messages\_pong\_sent:2185  cluster\_stats\_messages\_sent:4280  cluster\_stats\_messages\_ping\_received:2185  cluster\_stats\_messages\_pong\_received:2091  cluster\_stats\_messages\_received:427 |

7、CLUSTER info结点信息。这些信息会放到机器的配置中。

|  |
| --- |
| #结点id,机器的ip+port 集群总线端口,结点角色-结点状态(从节点会跟主节点id),节点最后一次发送ping命令过了多长时间, 结点纪元配置，结点网络情况，节点的槽。  10a687e0202ce14e80e464cafa22e856bad761c2 192.168.205.129:6371@16371 myself,master - 0 1625489668000 1 connected 0-5460  99dd38b284803a1a88cc0b3e9bf071a324da1b26 192.168.205.130:6375@16375 master - 0 1625489666023 5 connected 10923-16383  937e6dc769ea8eebcb0ff1d0539d0a162110d56d 192.168.205.130:6376@16376 slave db051f5af755cf80bb917d2594c158465d267c6b 0 1625489668038 3 connected  db051f5af755cf80bb917d2594c158465d267c6b 192.168.205.128:6373@16373 master - 0 1625489669044 3 connected 5461-10922  e0bc1a83bcabe508e1dd070439976830845fa6e4 192.168.205.129:6372@16372 slave 99dd38b284803a1a88cc0b3e9bf071a324da1b26 0 1625489670052 5 connected  318e405310e382ec8bf54c748e62b98502410f3c 192.168.205.128:6374@16374 slave 10a687e0202ce14e80e464cafa22e856bad761c2 0 1625489667030 1 connected |

## 集群的扩容

## 集群测试

读写时会对键值进行hash运算。进行分槽。

Set username zhagnsan

|  |
| --- |
| 192.168.205.129:6371> set username zhangsan  //这里做了转向。  -> Redirected to slot [14315] located at 192.168.205.130:6375  OK |

Get username

|  |
| --- |
| 192.168.205.129:6371> get username  //获取页做了转向。  -> Redirected to slot [14315] located at 192.168.205.130:6375  "zhangsan" |

## 总结

哨兵集群实现故障自动转移，但是当数据量过大导致生成 RDB 时间过长。而 Fork 执行的时候会阻塞主线程，由于数据量过大导致阻塞主线程过长，所以出现了 Redis 响应慢的表象。

使用 Redis Cluster 集群，主要解决了大数据量存储导致的各种慢问题，同时也便于横向拓展。在面向百万、千万级别的用户规模时，横向扩展的 Redis 切片集群会是一个非常好的选择。

集群的整个数据库被分为 16384 个槽（slot），数据库中的每个键都属于这 16384 个槽的其中一个，集群中的每个节点可以处理 0 个或最多 16384 个槽。

Redis 集群节点采用 Gossip 协议来广播自己的状态以及自己对整个集群认知的改变。

客户端连接到集群候任何一个实例后，实例会将哈希槽与实例映射信息发送给客户端，客户端将信息保存，用于将 key 定位到对应的节点。

集群并不能无限增加，由于集群通过 Gossip协议传播集群实例信息，所以通信频率是限制集群大小的主要原因，主要可以通过修改 cluster-node-timeout调整频率。